High Level Assembler Plugin
Project documentation

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

Supervisor: Miroslav Kratochvíl

Consultant: Slavomír Kučera (Broadcom)

Contents

1. Introduction

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.

We introduce such plugin that improves 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.

This document serves as an in-depth documentation for anyone who would like to understand the implementation of the project and the reasons behind it. It is advised that the potential contributors to the project read this documentation first.

1.1 Organisation of this document

First of all, in chapter 2, we briefly explain the basics of HLASM needed to comprehend the workflow of this language. In chapter 3, we provide an overview of the project's architecture, naming the most important components and indicating their relations. Then, we describe these components in separate chapters in further detail. In chapter 4, we state the responsibilities of the language server as the communication provider between the extension client and the parsing library. The workspace manager is the entry point to the parsing library used by the language server and it is fully described in chapter 5. The purpose of its sub-components is to handle file management, dependency resolution and parsing. The core of the processing of a HLASM file is implemented inside the analyzer, whose mechanics and implementation details are discussed in chapter 6. The project also provides macro tracing through the standard debugging procedure and it is fully explained in chapter 7.The last mentioned component, detailed in chapter 8 is the VSCode extension, which communicates with the language server and provides IDE features to the user. At the end of this document, in chapter 10, we provide the instructions on how to build the project.

Part I

Project overview

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*, which are used to produce both compile-time code and run-time code (see section 2.2). A statement consists of four fields separated by spaces that can be split into more lines using continuations (see section 2.1.2). Following are the existing fields:

- **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 to write a statement longer than 80 characters.

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

2.1.2 Continuations

Individual statements sometimes contain more than 80 characters, which does not agree with the historical line length limitations. Therefore, a special feature called *continuation* exists.

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 its right 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. This allows the presence of a continuation character even when the contents of a line have not reached the continuation column (see listing 3).

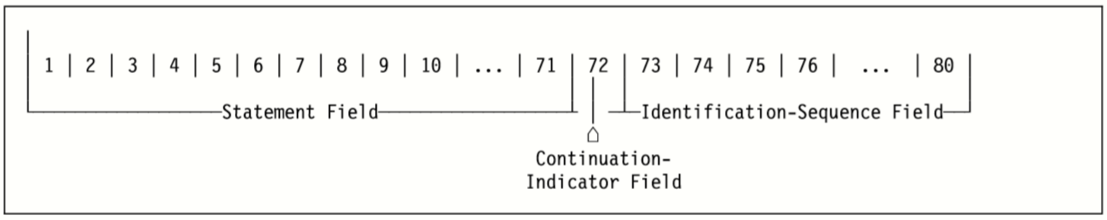

Figure 2.1: Description of line columns (source: HLASM Language Reference).

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 outlined the syntax, we now describe the assembly process of HLASM.
We distinguish two types of processing:

- *conditional assembly (CA) processing* — the main purpose of which is to generate statements for ordinary assembly (see section 2.2.3)
- *ordinary assembly processing* — which handles *machine instructions* and *assembler instructions* (see section 2.2.1.2, section 2.2.1.3)

2.2.1 Ordinary assembly

Ordinary assembly, along with machine and assembler instructions, is responsible for the runtime behavior of the program. It allows the generation of code from both traditional machine instructions and special-purpose assembler instructions. Moreover, it assigns values to *ordinary symbols*.

2.2.1.1 Ordinary symbols

In HLASM, an *ordinary symbol* is a named run-time constant. It is defined by inputting its name into the name field of a statement along with a special assembler instruction. Each ordinary symbol can only be defined once, and its value is constant. There are two types of ordinary symbols:

- An *absolute symbol* that simply has an integral value.
- A *relocatable symbol* that represents an address in the resulting object code. A relocatable symbol can also be defined by writing the ordinary symbol name into the name field of a statement along with a machine instruction name. The symbol then denotes the address of the given instruction.

In addition to symbol value, ordinary symbols also contain a set of *attributes*, the most common ones being *type* and *length*.

2.2.1.2 Machine instructions

Machine instructions represent the actual processor instructions executed during runtime. Similarly to traditional assemblers, they are translated into corresponding opcodes and their operands are processed. However, HLASM also allows expressions to be passed as their operands, which may use ordinary symbols and support integer and address arithmetic.

Listing 4 Examples of data definition

```
CL8'ABC'
    C is type of data definition - array of characters
    L8 specifies length - 8 bytes
    'ABC' is nominal value of the data definition
    CL8'ABC' would assemble 8 bytes, first three of which would be EBCDIC
             representations of letters A, B and C

5AY(A+4,B)
    5 is duplication factor - the nominal value will be repeated 5 times
    A is type - address
    Y is type extension - it modifies the length of the address
    (A+4,B) is nominal value - comma separated expressions that will
            be assembled as addresses
    5AY(A+4,B) would assemble total of 20 bytes, first 2 bytes is value
               of expression A+4, then B and then 4 more copies of the same
```

2.2.1.3 Assembler instructions

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

Following are examples of assembler instructions:

- **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.). These instructions take *data definition* as operands. Listing 4 shows examples of data definition.
- **EQU** — Defines ordinary symbols.
- **COPY** — Copies text from a specified file[1] (called *copy member*) and pastes it in place of the instruction. It is very similar to the C preprocessor `#include` directive.
- **CSECT** — Creates an executable control section, which serves as the start of relative addressing. It is followed by sequence of machine instructions.

2.2.1.4 Ordinary symbols resolution

All the assembler instructions and ordinary symbols must be resolved before the assembler creates the final object file. However, as the HLASM language supports forward

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

Listing 5 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
```

declaration of ordinary symbols, the assembly may be quite complicated. Consider an example in listing 5. When the instruction on 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 on 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. However, 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 from these principles can be arbitrarily deep and complicated, however it must not contain cycles (a symbol must not be transitively dependent on itself).

2.2.2 Object file layout

The product of ordinary assembly is an object file. Let us briefly describe its layout.

2.2.2.1 Sections

An object file consists of so-called *sections*. They are user-defined (by instructions CSECT, DSECT, ...) and can be of different kinds, each with various properties. Absolute positions of sections within the object file are undefined — they are determined automatically after the compilation. This also implies that all relocatable symbols are only defined relatively to the section that contains them.

2.2.2.2 Location counter

Any time a machine instruction is encountered, its opcode is outputted to the *next available address*. Each section has a structure pointing to this address — a so-called *location counter*.

The user may define more location counters and then arbitrarily switch between them to state the next address for code generation. Therefore, at all times, there is one and

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

only one location counter active, which defines where the next machine instruction will be generated.

At the end of assembly, all code denoted by location counters is assembled in a well-defined order, and so the absolute position of all relocatable symbols within their section is known.

The value of the location counter can be arbitrarily changed by the ORG instruction. It can be moved backwards or forwards (with restriction of counter underflow) to set the next address. This means that user can generate some code, move counter backwards and overwrite it. Then the ORG instruction can be used to set location counter to the next available untouched address to continue in object creation.

2.2.3 Conditional assembly

Conditional assembly is another feature provided by HLASM. It is essentially a macro-language built on top of a traditional assembler.

User may use conditional assembly instructions to either define *variable symbols*, which can be used in any statement to alter its meaning, or to define *macros* — reusable pieces of code with parameters. Based on these instructions, conditional assembly then alters the textual representation of the source code and selects which lines will be processed next.

2.2.3.1 Variable symbols

‘ Variable symbols serve as compile-time variables. Statements that contain them are called *model statements*.

During conditional assembly, variable symbols are substituted for their value to create a statement processable by ordinary assembly. For example, a user can write a variable symbol in the operation field 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 integer, boolean or string. CA instructions gather this information for different sorts of conditional branching.

2.2.3.2 CA instructions

CA instructions are not assembled into object code. They are used to select which instructions will be processed by the assembler next.

One example of their capabilities is conditional and unconditional branching. As HLASM provides a variety of built-in binary or unary operations on variable symbols, complex conditional expressions can be created. This is important in HLASM, as the user can alter the flow of instructions that will be assembled into an executable program.

Another subset of CA instructions operates on variable symbols. These can be used to define variable symbols locally or globally, assign or update their values.

2.2.3.3 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 also contain CA instructions, or even a sequence of instructions generating another macro definition. With the help of variable symbols, HLASM has the power to create custom, task specific macros.

An example of a simple HLASM program with the description of its statements is shown in listing 6.

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

On 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`.

On 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 next line, there is a CA branching instruction `AIF`. If value of `VAR` equals 4, all the text between `AIF` and `.END` is completely skipped and assembling continues on line `18`, where the branching symbol `.END` is located.

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

On 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` on line `17` creates value of type double and assigns its address to the 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 text preprocessing, it is still interlinked with ordinary processing. CA has mechanics that allow the assembler to gather information about statements that are printed during the processing. It can also access values created in ordinary assembly and use them in conditional branching, and is able to lookup constants that are not yet defined prior to the currently processed statement. During ordinary assembly, names of these instructions can also be aliased.

To sum up, CA processing has variables for storing values during the compilation and CA instructions for conditional branching. Hence, it is Turing-complete while still evaluated during compile-time.

2.3 HLASM source structure

The file that generates the object code is called an *open-code* file. It is the entry file of the HLASM compiler. Each open-code file can have in-file dependencies, specifically:

- External Macro definitions
- Copy members

Listing 6 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
```

These are not treated as open-code files because they do not directly generate object code. Rather, they serve as statement sequences that are included in specific places of open-code and provide specific meaning.

3. Architecture overview

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. 3.1 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 the Go to definition command) are transformed into standard LSP messages and are 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 implemented the LSP language server and written a lightweight extension to an editor, which uses an already existing implementation of the LSP client. To implement source code highlighting, we had to extend the protocol with a new notification. This notification is used for transferring information from the language server to the VS Code client, which is extended to highlight code in editor based on the incoming custom notifications.

In this chapter, 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. An overview of the architecture is pictured in fig. 3.2. The architecture of whole project is shown in appendix A

3.1 Language server component

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 either a 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
- asynchronous request handling: when a user makes several consecutive changes to a source code, parsing on every change is not needed

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

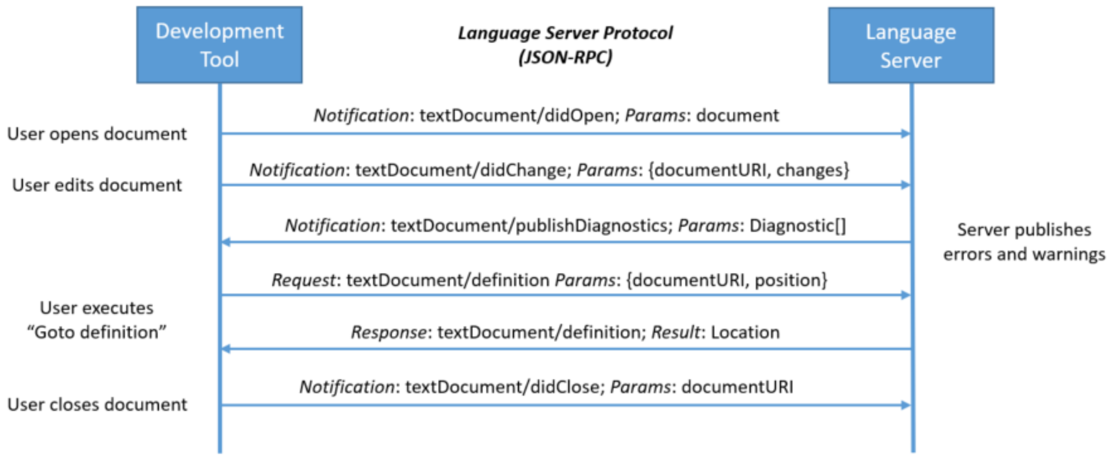
Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
Notification: textDocument/didClose; Params: documentURI
User closes document

Figure 3.1: LSP session example. (source: `https://microsoft.github.io/language-server-protocol/overview`)

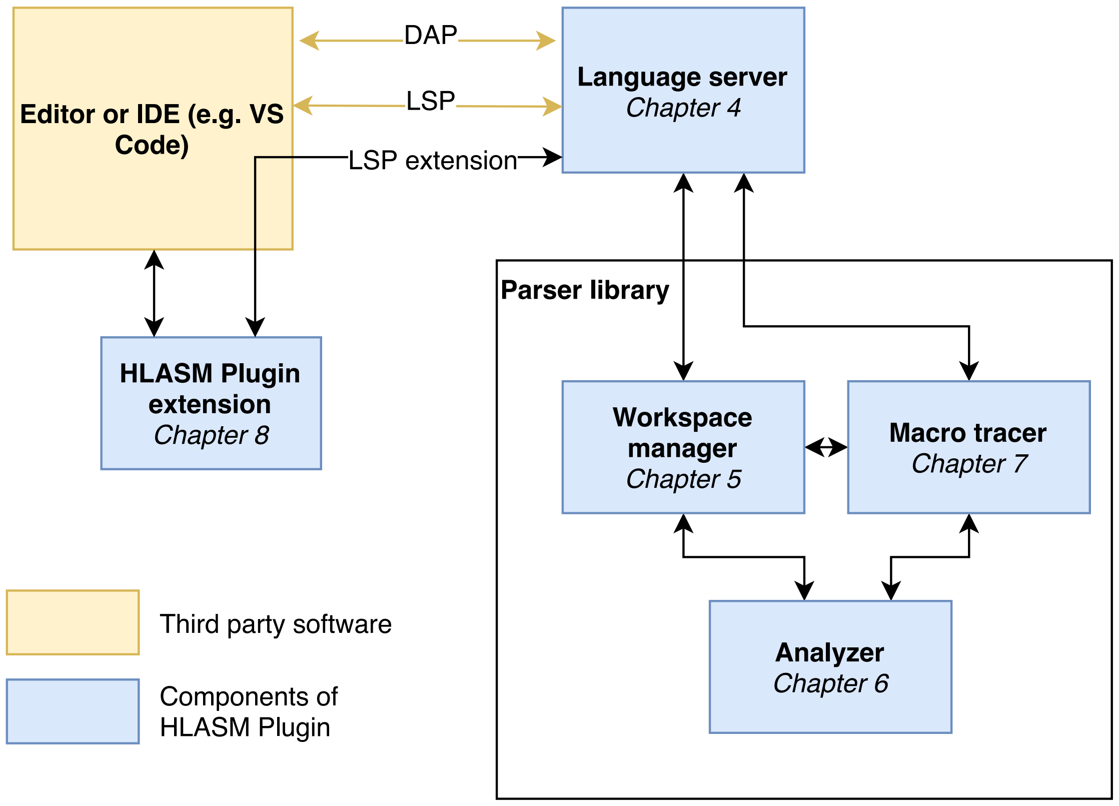
Editor or IDE (e.g. VS Code)
DAP
LSP
LSP extension
Language server Chapter 4
HLASM Plugin extension Chapter 8
Parser library
Workspace manager Chapter 5
Macro tracer Chapter 7
Analyzer Chapter 6
Third party software
Components of HLASM Plugin

Figure 3.2: The architecture of HLASM Plugin

3.2 Parser library component

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.

3.2.1 Parser library API

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

At first, the API implements 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 that 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 has the exact contents of all files in open 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`) that inform the library about files that are currently open in the editor and their exact contents.
- `DidChangeWorkspaceFolders` notification that informs the library when a workspace has been opened or closed.
- `DidChangeWatchedFiles` notification

Next, the API implements the requests and notifications that provide the parsing results, specifically:

- `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.

3.2.2 Analyzer

The analyzer processes a single HLASM file. 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 analysis of HLASM code includes:

- recognition of statements and their parts (lexing and parsing)
- interpretation of instructions that should be executed in compile time
- reporting of problems with the source by producing LSP diagnostics
- providing highlighting and LSP information

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

3.3 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 to create a connection between them.
- to implement extension of the LSP protocol for enabling 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 support for editing lines with continuations — when the user types something in front of the continuation character, it should stay in place.

3.4 Macro tracer

The macro tracer gives a possibility to trace the compilation of HLASM source code in a way similar to common debugging. This is the reason why we chose to implement support for the Debug Adapter Protocol [2] (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 main responsibilities of the macro tracer include:

- `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
- `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

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

- `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. It analyzes 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.

Part II

Component description

4. Language server

The purpose of the Language server is to implement the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP) and to provide access to the parser library by using them. It has to deserialize and serialize LSP and DAP messages, extract parameters of particular methods and then serve the requests by invoking functionality of parser library.

4.1 Language Server Protocol

Language Server Protocol is used to extend code editors with support for additional programming languages. LSP defines 2 communicating entities: a client and a server. The LSP client is editor-specific and wraps interaction with the user. The LSP server is language-specific and provides information about the source code.

The main purpose of the LSP is to allow the language server to provide language-specific response to various user interactions with the code editor. Messages that flow through LSP can be divided into three categories:

- **Parsing results presentation** Messages from the first category allow the language server to send results of source code analysis to the LSP client. The editor is then able to show them to the user. For example, when the user clicks on a symbol in HLASM code and then uses the 'Go to definition' function, the LSP client sends a request to the language server with the name of currently open file and current location in the file. The server is then expected to send back the location of the definition, so the editor can present it to the user (e.g. the editor moves the caret to the definition location). List of all such messages is in table 4.1.
- **Editor state and file content synchronization** Messages from the second category flow mainly from the client to the server and ensure that the server has enough information to correctly analyze source code. List of all such messages can be found in table 4.2.
- **LSP initialization and finalization** Lastly, there are several messages that handle protocol initialization and finalization.

LSP is based on JSON RPC[1]. There are two types of interaction in JSON RPC: requests and notifications. Both of them carry the information to invoke a method on the recipient side — name of the method and its arguments. The difference between the two is

[1]`https://www.jsonrpc.org/specification`

Message	Description
textDocument/definition	The client sends a position in an open file. The server responds with a position of a definition of a symbol at that position.
textDocument/references	The client sends a position in an open file. If there is a symbol, the server responds with a list of positions where the symbol is used.
textDocument/hover	The client sends a position in an open file where the user is pointing with the cursor. The server responds with a string to be shown in a tooltip window.
textDocument/completion	The client sends a position in an open file and how a completion box was triggered (i.e. with what key, automatically/manually). The server responds with a list of strings suggested for completion at the position.
textDocument/ publishDiagnostics	The server sends diagnostics to the client. A diagnostic represents a problem with the source code, e.g. compilation errors and warnings.

Table 4.1: List of all results-presenting messages

Message	Description
textDocument/didOpen textDocument/didChange textDocument/didClose	The server is notified whenever the user opens a file, changes contents of an already open file or closes a file in the editor.
workspace/ didChangeWatchedFiles	The client notifies the server when a watched file is changed outside of the editor. Watched files selector is defined when the client is started (in the extension component).
workspace/ didChangeWorkspaceFolders	The client notifies the server that the user has opened or closed a workspace.

Table 4.2: List of all implemented editor state and text synchronization messages

Listing 7 An example of a message sent from the client to the server.

```
Content-Length: 123\r\n
\r\n
{"jsonrpc":"2.0","method":"textDocument/didClose","params":{"textDocument":
{"uri":"file:/c%3A/Users/admin/Documents/source.hlasm"}}}
```

that each request requires a response containing the result of the method, whereas the notifications do not.

The LSP uses the JSON RPC specification and further specifies how messages are transferred and defines methods, their arguments, responses and semantics. A raw message sent from the client to the server is shown in listing 7.

The raw messages have HTTP-like headers. The only mandatory header is `Content-Length`, which tells the recipient the length of the following message. The JSON itself is sent after the header.

Inside the JSON, there is a name of the method to be invoked and parameters to pass to the method. In this case, the client is sending a notification that file `C:/Users/admin/Documents/source.hlasm` was closed in the editor by the user. As it is a notification, there must not be any response.

On top of this basic protocol, LSP defines methods and their semantics to cover common functionality that users expect when programming in an editor. List of all methods implemented in the language server can be found in table 4.3.

4.2 DAP

Debug Adapter Protocol is used to extend code editors with debugging support for additional programming languages. We use it to provide the user with the ability to trace how the HLASM compiler processes source code step by step. The user can see the values of compile-time variables and follow the expansion of macros in debug-like experience.

The communication in DAP is between an editor or an IDE and a debugger. The editor notifies the debugger about the user actions, e.g. when a breakpoint is set or when the user uses step in/step over buttons. The debugger informs the editor about the state of the debugged application, for example when the debugger stopped because it hit a breakpoint. While it is stopped, the debugger sends information about program stack, variables valid in current debugger scope and its values.

DAP is very similar to LSP. Although the ideas behind DAP are nearly the same, DAP is not based on the JSON RPC. Instead, DAP specifies its own implementation of remote procedure call, still using JSON as the basic carrier of the messages. DAP has requests and events — requests always go from the client to the server and require response. Events are the same as the notifications from JSON RPC that are sent from the server to the client. The similarity allows our language server component to share a lot of code between the implementations of the protocols.

4.3 Language server overview

The architecture of the Language server component is illustrated in fig. 4.1. It communicates on the standard input/output via LSP with the LSP client and listens on a TCP port to provide DAP support for the macro tracer. The TCP communication is wrapped by class `tcp_handler`, which abstracts from the complexity of communicating through TCP/IP.

The main purpose of the class `dispatcher` is to provide abstraction for the lowest level communication, which is shared by LSP and DAP. It reads iostream to parse messages using the JSON for Modern C++ library (see chapter 9) and stores them in the `request_manager` as `requests`.

A `request` encapsulates one message that came from the client and is basically represented only by raw (but parsed) JSON.

`request_manager` stores `requests` in a queue and runs a worker thread that serves the requests one by one. As there is only one instance of `request_manager` running in the language server, it serializes requests from DAP and LSP (which come asynchronously from separate sources) into one queue.

`server` is an abstract class that implements protocol behavior that is common for both DAP and LSP — it basically implements Remote Procedure Call. Actual handling of LSP and DAP requests is implemented in `features`. Each `feature` contains implementation of several protocol requests or notifications. The features unwrap the arguments from JSON and call corresponding parser library methods.

There are two implementations of the abstract `server` class: `lsp_server` and `dap_server`. They both implement the initialization and finalization of protocol communication, which is a bit different for both protocols and both use features to serve protocol requests.

4.4 Example: hover request handling

The fig. 4.2 shows handling of the hover request in the language server. The hover request is sent from the LSP client to the `lsp_server` when the user hovers over the text of a file. The hover request contains location of the mouse cursor in text, i.e. the name of the file, the number of line and column where the cursor is. The LSP client then expects a response containing a string (possibly written in markdown language) to be shown in a tooltip box.

The whole process begins with reading from the standard input by the LSP instance of the `dispatcher`. It first reads the header of the message, which contains the information about the length of the following JSON. Then it reads the JSON itself and deserializes it using the JSON for Modern C++ library (see chapter 9). All other components of the language server work only with the parsed representation of the message. The `dispatcher` adds the message to the `request_manager` and returns to reading the next message from the standard input.

The request in the `request_manager` either waits in a queue to be processed, or, if the queue was empty, the worker thread is woken up from sleep using conditional variable. The worker then passes the JSON to the `lsp_server`, which looks at the name of the method written in the message and calls the method "hover" from the language feature.

The hover method unpacks the actual arguments from JSON and converts any URIs to paths using the cpp-netlib URI library. Then, it calls the hover method from the parser

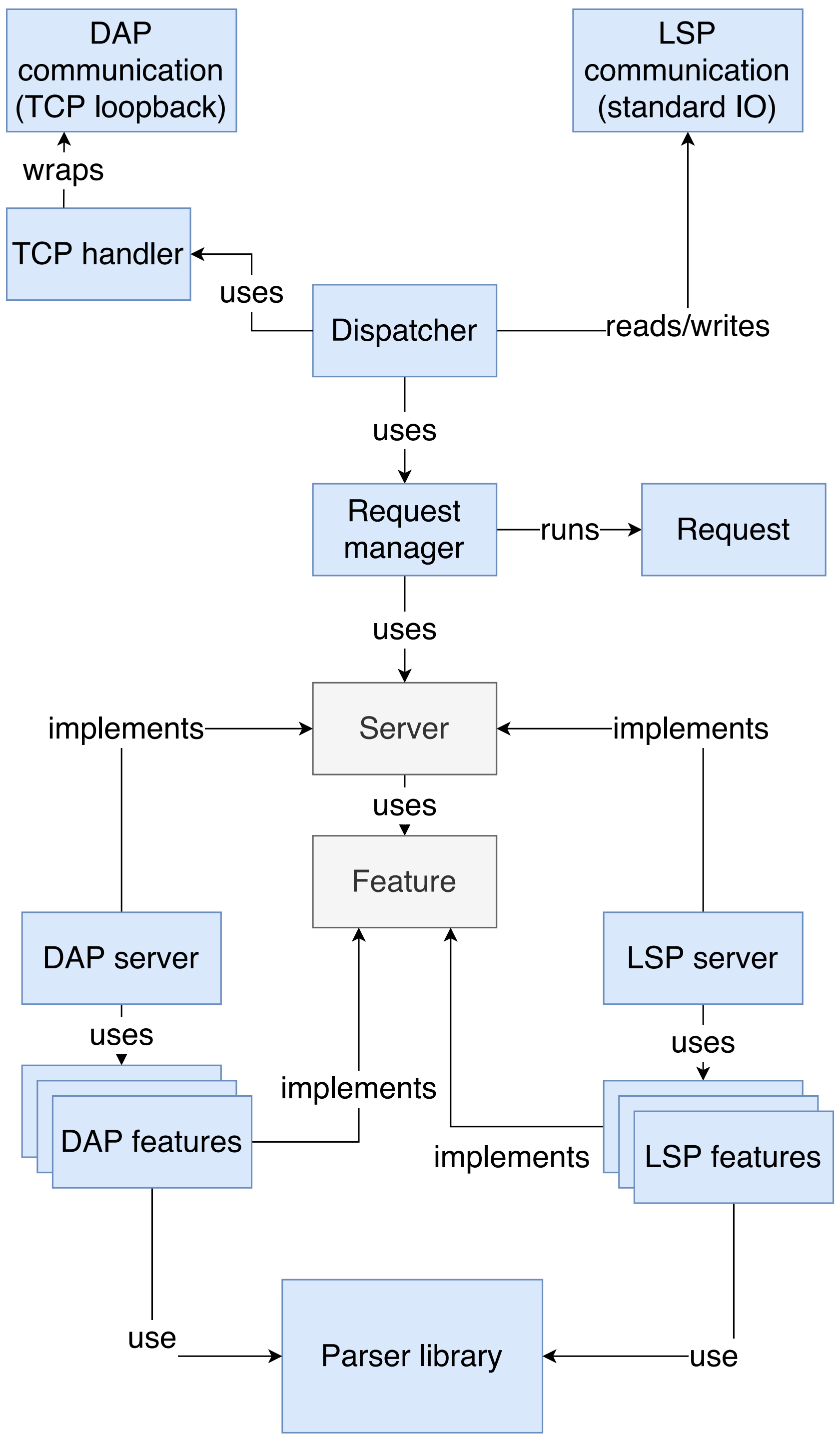

Figure 4.1: Architecture of language server.

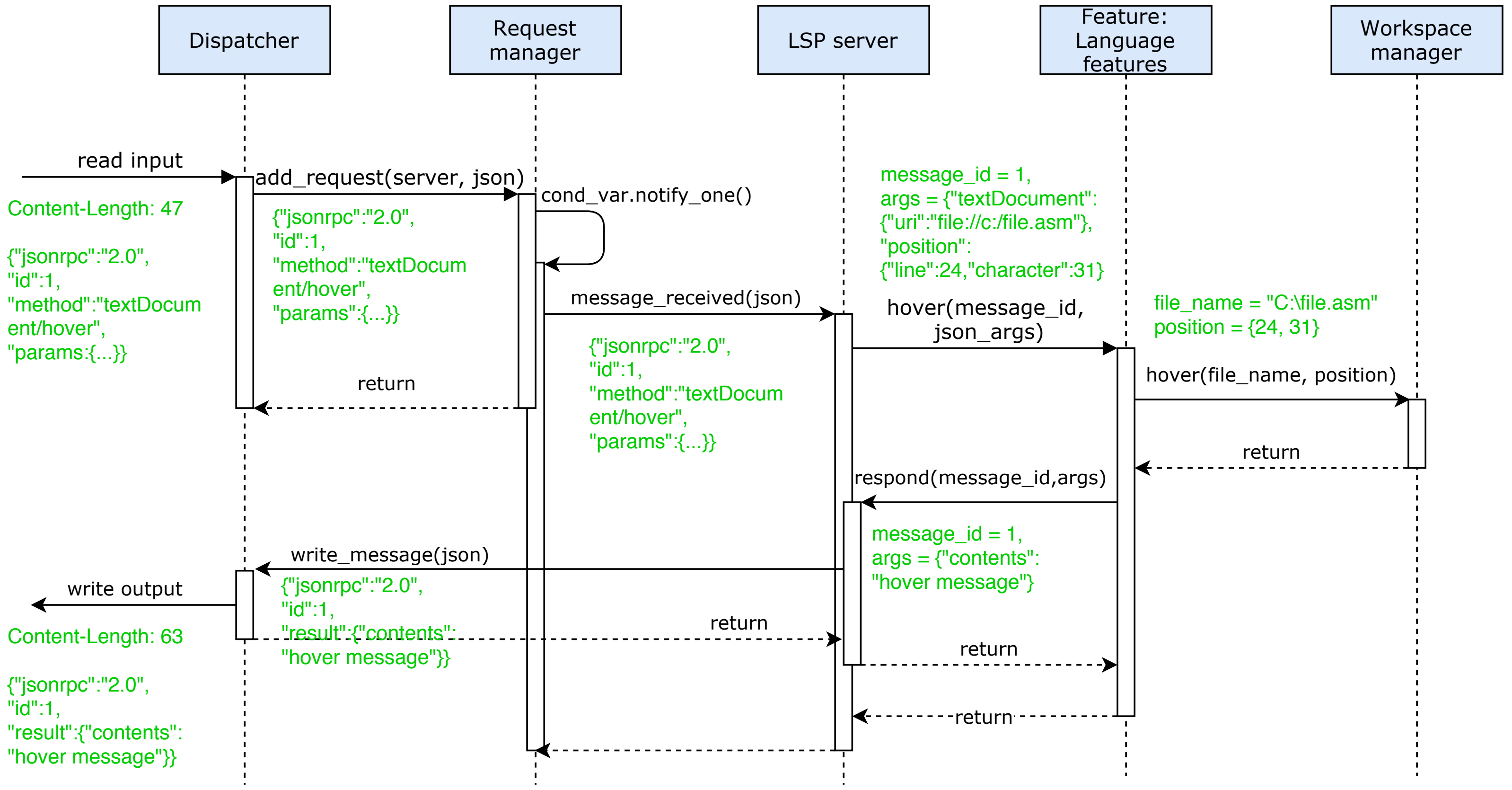
Dispatcher
Request manager
LSP server
Feature: Language features
Workspace manager
read input
Content-Length: 47
{"jsonrpc":"2.0", "id":1, "method":"textDocument/hover", "params:{...}}
add_request(server, json)
{"jsonrpc":"2.0", "id":1, "method":"textDocument/hover", "params":{...}}
cond_var.notify_one()
return
message_received(json)
{"jsonrpc":"2.0", "id":1, "method":"textDocument/hover", "params":{...}}
message_id = 1, args = {"textDocument": {"uri":"file://c:/file.asm"}, "position": {"line":24,"character":31}
hover(message_id, json_args)
file_name = "C:\file.asm" position = {24, 31}
hover(file_name, position)
return
respond(message_id,args)
message_id = 1, args = {"contents": "hover message"}
write_message(json)
{"jsonrpc":"2.0", "id":1, "result":{"contents": "hover message"}}
write output
Content-Length: 63
{"jsonrpc":"2.0", "id":1, "result":{"contents": "hover message"}}
return
return
return

Figure 4.2: A sequence diagram showing the processing of the textDocument/hover request in the language server. The green text represents an example of data passed in the arguments.

library, which returns a string to be shown in the tooltip next to the hovering mouse. The language feature then wraps the return value back in JSON and calls the `respond` method of its `response_provider` implemented by the `lsp_server`.

The `lsp_server` wraps JSON arguments into a LSP response and uses the `send message provider` implemented by `dispatcher` to send it to the LSP client. The `dispatcher` serializes the JSON, adds the header with the length of the JSON and writes the message to a standard output. Finally, all methods return and the worker thread in `request_manager` looks for another request. If there is none, it goes to sleep.

4.5 I/O handling

`language_server/src/main.cpp`
`language_server/src/dispatcher.h`
`language_server/src/dap/tcp_handler.h`

The purpose of the `dispatcher` is to abstract from the complexity of working with raw strings and streams. It executes an infinite loop in which it reads messages from `std::iostream` and adds them to the `request_manager` as parsed JSON objects. At the same time, it is able to write responses in the correct format.

The language server communicates with the LSP client on a standard input and output, so we simply use the `dispatcher` with the standard `std::cin` and `std::cout` objects to communicate with the LSP client.

The DAP communicates using TCP/IP, which is less straightforward. Before the VS Code extension starts the language server, it finds a free TCP port and passes it as an argument to the language server executable. The `TCP handler` then starts listening on that port. Once the user wants to start the macro tracer, the DAP client connects to the port on localhost. The `tcp_handler` accepts the TCP client and creates a `dispatcher` and a `dap_server`. Once the DAP communication ends, both the `dispatcher` and the `dap_server` are destroyed and the `tcp_handler` starts listening again for the next DAP session. Thanks to the ASIO library (see chapter 9) implementation of the `std::iostream` interface, the `dispatcher` is able to completely abstract from the fact that it is communicating through TCP and not through the standard IO.

4.6 LSP and DAP Server

`language_server/src/server.h`
`language_server/src/feature.h`
`language_server/src/lsp/lsp_server.h`
`language_server/src/dap/dap_server.h`
`language_server/src/dap/dap_feature.h`

The servers are able to process incoming LSP and DAP requests. They get the messages in a form of already parsed JSONs. Then they extract the name of the requested method with its parameters from the message and call the corresponding method with the parameters encoded as JSON.

There are two server implementations: `lsp_server` and `dap_server`. Both inherit from an abstract class called `server`. They implement protocol-specific processing of messages — although the protocols are quite similar (both are based on RPC), each protocol has different initialization and finalization, different message format, etc.

The functionality of servers is divided into `features`. Each feature implements several LSP or DAP methods by unpacking the arguments of the respective method and calling corresponding parser library function. During initialization, each feature adds its methods to the server's list of implemented methods. The `lsp_server` uses three features:

Component	LSP Method name
LSP server `language_server/src/lsp/lsp_server.h`	initialize shutdown exit textDocument/publishDiagnostics
Text synchronization feature `language_server/src/lsp/feature_text_synchronization.h`	textDocument/didOpen textDocument/didChange textDocument/didClose textDocument/semanticHighlighting
Workspace folders feature `language_server/src/lsp/feature_workspace_folders.h`	workspace/didChangeWorkspaceFolders workspace/didChangeWatchedFiles
Language feature `language_server/src/lsp/feature_language_features.h`	textDocument/definition textDocument/references textDocument/hover textDocument/completion

Table 4.3: The list of all implemented LSP methods and the classes where they are implemented

- *Text synchronization feature*, which handles the notifications about the state of open files in the editor.
- *Workspace folders feature*, which handles the notifications about adding and removing workspaces.
- *Language feature*, which handles requests about HLASM code information.

The table 4.3 shows the list of all implemented LSP methods and the classes where the implementations lie.

The DAP server uses only one feature — the Launch feature, which handles stepping through the code and retrieving information about both variables and stack trace. The table table 4.4 shows the list of all implemented DAP methods.

4.7 Response with result

`language_server/src/feature.h`
`language_server/src/server.h`

According to the LSP and the DAP, the server is required to send messages back to the LSP/DAP client either as responses to requests (e.g. `hover`), notifications (e.g. textDocument/publishDiagnostics notification) or events (e.g. stopped event). Features require reference to an instance of the `response_provider` interface that provides methods `respond` and `notify` for sending messages back to the LSP client. Both LSP and DAP server classes implement the `response_provider` to form protocol-specific JSON with the arguments.

The servers then send the JSON to the LSP/DAP client using the `send_message_provider` interface. At this point, the final complete JSON response is formed. The `send_message_provider` then adds the message header and serializes the JSON using the JSON for Modern C++ library (see chapter 9).

The only implementation of the `send_message_provider` interface is the `dispatcher`.

Class	Source file	DAP Method name
dap_server	language_server/src/-dap/dap_server.h	initialize disconnect
launch_feature	language_server/src/-dap/feature_launch.h	launch setBreakpoints configurationDone threads stackTrace scopes next stepIn variables continue stopped exited terminated

Table 4.4: The list of all implemented DAP methods and the classes where they are implemented

4.8 Request Manager

language_server/src/request_manager.h

`request_manager` encapsulates a queue of requests with a worker thread that processes them. There may be up to two `dispatcher` instances in the language server: one for LSP and one for DAP. Both of them add the requests they parse into one `request_manager`. It is necessary to process the requests one by one, because the parser library cannot process more requests at the same time.

Asynchronous communication is handled by separating the communication into threads:

- LSP read thread — a thread in which the `dispatcher` reads messages from the standard input.
- DAP read thread — a thread in which the `tcp_handler` listens on a localhost port to initiate a DAP session. After accepting the DAP client, the `dispatcher` reads DAP input on this thread too.
- Worker thread in `request_manager` that processes each request using the `lsp_server` or the `dap_server` and ultimately the parser library.

The threads are synchronized in two ways: First, there is a mutex that prevents the LSP and the DAP threads from adding to the request queue simultaneously. Second, there is a conditional variable to control the worker thread.

`request_manager` additionally incorporates a mechanism for invalidating requests that have been obsoleted by new requests. The obsoleting of requests is done by a cancellation token. It is shared between the parser library and the `request_manager`. When set to true, the results of current request or notification are no longer needed, the parser library stops all parsing and return as soon as possible.

When a new request arrives, all previous requests (including the currently processed one) that concern the same file are invalidated. However, they cannot be simply removed from the queue. They still have to be processed as they may carry information that must not be discarded (e.g. changes to contents of a file). The parser library processes the request but does not reparse any source files.

4.8.1 Example of request invalidating

For example, when a user starts changing a file, every character he writes is passed to the language server as a textDocument/didChange notification. Each such notification is processed in two stages:

1. The parser library changes the internal representation of the text document.
2. The parser library starts the parsing of the file to update diagnostics and highlighting. This may take some time.

When more didChange notifications come in succession, their first parts must be executed with all the notifications to keep the internal representation of the file updated. However, the user is interested only in diagnostics and semantic highlighting for the current state of the text, so we need to parse the file only once — after the last notification.

5. Workspace Manager

Workspace manager encapsulates all functionality of the parser library. It is the access point to all parsing capabilities, keeps the current state of all open files and resolves libraries needed by the analyzer. It also manages when files should be reparsed.

5.1 Parser library API

```
parser_library/include/protocol.h
parser_library/include/range.h
parser_library/include/workspace_manager.h
parser_library/src/workspace_manager_impl.h
```

First of all, the workspace manager component is the only public interface of the parser library. The API design is based on LSP and DAP, most of the API is just LSP/DAP rewritten in C++. The API uses the observer pattern for DAP events and notifications originating in parser library (e.g. textDocument/publishDiagnostics).

The API methods can be divided into three categories:

- Editor state and file content synchronization (table 5.1)
- Parsing results presentation
- Macro tracer

Method	Description

`did_open (file name, file content)`
`did_change (file name, changes)`
`did_close (file name)`
	Three methods that are called whenever the user opens a file, changes contents of an already opened file or closes a file in the editor.

`did_change_watched_files (file paths)`	Method, that is called when a file from a workspace has been changed outsize of the editor

`add_workspace (ws name, ws path)`
`remove_workspace (ws path)`
	Methods that are called when the user opens or closes a workspace in the editor

Table 5.1: List of all Editor state and text synchronization methods

Method	Description
`definition(file name, caret position)`	The method gets a position in an opened file. If there is a symbol, the method returns position of definition of that symbol
`references(file name, caret position)`	The method gets a position in an opened file. If there is a symbol, the method returns list of positions where the symbol is used
`hover(file name, mouse position)`	The method gets a position in an opened file where the user points with cursor. Returns list of strings to be shown in a tooltip window
`completion(file name, mouse position, trigger info)`	The method gets a position in an opened file and how the completion box was triggered (i. e. with what key, automatically/manually). Returns list of strings suggested for completion at the position

Table 5.2: List of all parse results methods

5.1.1 Editor state and file content synchronization

All the methods from the first category are listed in table 5.1. There are two types of files that need to be synchronised:

- Files, that the user has opened in the editor. Those files are being edited by the user and their content may be different from the files actually saved in the filesystem.
- Files, that the parser library opens from the hard disk, because they are needed to parse opened files (e.g. a macro that is used by an opened file)

So the parser library is allowed to load arbitrary files from the disk, and use its contents until such file is opened in the editor. From that point on, the only source of truth for the contents of the file are the did_change notifications. Once the file is closed in the editor, the parser library is again allowed to rely on its contents in the filesystem.

5.1.2 Parsing results presentation

All the methods from the second category are listed in table 5.2. They get position of caret or mouse cursor in a file and are expected to return information about the place in the code. For example, method `hover` is called when the user points at some word in the code and waits for a short time. The method returns a string that the editor shows in the tooltip window at the position. Typically, the tooltip would show type of the variable and its value, if known.

Additionally, the parser library presents its results using the observer pattern. There are two interfaces: highligting and diagnostics consumer. Each of them has method `consume` that gets updated information as parameter whenever there is an update. Any potential user of the library (e.g. the language server component) just has to implement the interfaces to process the results.

5.1.3 Macro tracer

The macro tracer part of the API is again just DAP rewritten in C++. There are methods that are called when the user clicks on buttons to control the macro tracer: launch the tracer, step in, step over, continue and stop. Moreover, there are methods that retrieve information about current state of traced code: stack of macro calls and information about compile time variables. See chapter 7 for full description of macro tracer.

5.2 Libraries configuration

The parser library approaches the dependency resolution in a way similar to the mainframe. On mainframe, you would have to define the locations of your dependencies in a JCL file [1]. As the user may want to include tens of dependencies for multiple open codes, a source code management tool called Endevor[2] groups these dependencies into so-called *processor groups*. Then, the user only has to assign a processor group to the open code and the Endevor does the dependency resolution for him.

To provide similar experience with local files, the parser library simulates this behavior. If the user wants to include dependencies in his project, he has to define 2 configuration files inside his workspace: *pgm_conf.json* and *proc_grps.json*. The workspace component of the parser library then processes the configurations, retrieving their values upon initialization. Moreover, each time a save command is issued on any configuration file, the configuration values are reloaded via `load_config` method.

5.2.1 Processor groups

The proc_grps configuration file contains a JSON array of possible processor groups, which consist of a name and an array of folder paths (may be relative to the root of the workspace). An example can be found in listing 8.

Whenever `load_config` is called, the workspace retrieves these processor groups from the configuration file and creates libraries. The libraries provide information about paths to their dependency files. During the parsing, the workspace retrieves the library corresponding to the provided processor group name and uses it to search for a wanted macro or copy file.

5.2.2 Program configuration

The pgm_conf configuration file contains a JSON array of program names (or wildcards 8.3.1), matched to their processor groups. It serves as a list of the HLASM open code files and states the libraries (in form of processor groups) that contain the dependencies of each open code. An example can be found in listing 9.

[1] `https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zjcl/zjclc_basicjclconcepts.htm`

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

Listing 8 A processor group configuration file

```
{
  "pgroups": [
    {
      "name":"GROUP1",
      "libs": [
        "ASMMAC/",
        "C:/SYS.ASMMAC"
      ]
    },
    {
      "name":"GROUP2",
      "libs": [
        "G2MAC/",
        "C:/SYS.ASMMAC"
      ]
    }
  ]
}
```

From this configuration, the workspace simply remembers the processor group - open code mapping.

5.3 Architecture overview

The architecture of the parser library is organized into the following components:

Workspace manager API The workspace manager provides API for handling various workspace management (e.g. add new workspace), LSP and DAP requests. It may hold multiple workspaces and calls file manager to handle changes in the workspace files.

Workspace representation The representation of workspace deals with the relations between its files (dependencies) upon parse request and propagates the parsing further into analyzer. It also retrieves data from the configuration files and it is used for resolving dependency searches by implementing parse library provider.

Processor group representation `parser_library/src/workspaces/processor_group.h`
The representation of a processor group uses the API of libraries to search for their dependencies. Currently, we only support local libraries, which utilize the file manager for their file information retrieval.

File manager `parser_library/src/workspaces/file_manager.h`
The file manager is used by multiple components to handle file management and file searches. It also distinguishes and does conversions between regular files and processor files, which may be used for parsing.

Listing 9 A program configuration file

```
{
  "pgms": [
    {
      "program": "source_code",
      "pgroup": "GROUP1"
    },
    {
      "program": "second_file",
      "pgroup": "GROUP2"
    },
  ]
}
```

Analyzer The analyzer accepts a file along with the information needed for dependency resolution, syntactically and semantically processes it and fills the context tables. The component is further explained in chapter 6.

An overview of the architecture is visualised in fig. 5.1.

The technical details of each component are further explained in the following sections.

5.4 Workspace representation

`parser_library/src/workspaces/workspace.h`

The representation of workspace is used by the workspace manager to handle various changes in the workspace. The workspace manager propagates LSP requests and notifications coming from the language server to the corresponding workspace and retrieves the results from it via the registered observers.

The workspace component uses the file manager for the file searches, retrieves the values from the configuration files and creates processor groups and is capable of resolving dependencies.

Due to the possibility to include files, the workspace maintains a list of dependants, which are active dependencies of another workspace files. The list of dependants is needed, for example, in case the user changes contents of a macro that is used by multiple open code files, as all of them would have to be reparsed.

The core of the workspace is its `parse_file` method. As addition to the parsing part, it also ensures that the file to be parsed, its dependencies and dependants provide consistent results. The method works as follows:

1. It checks whether the parsed file is a configuration file. If so, the workspace reloads the configuration values and reparses all dependants in the workspace.
2. In case the parsed file is not a configuration file, it creates a list of all files to be parsed. This list consists of files depending on the parsed file and the parsed file itself.

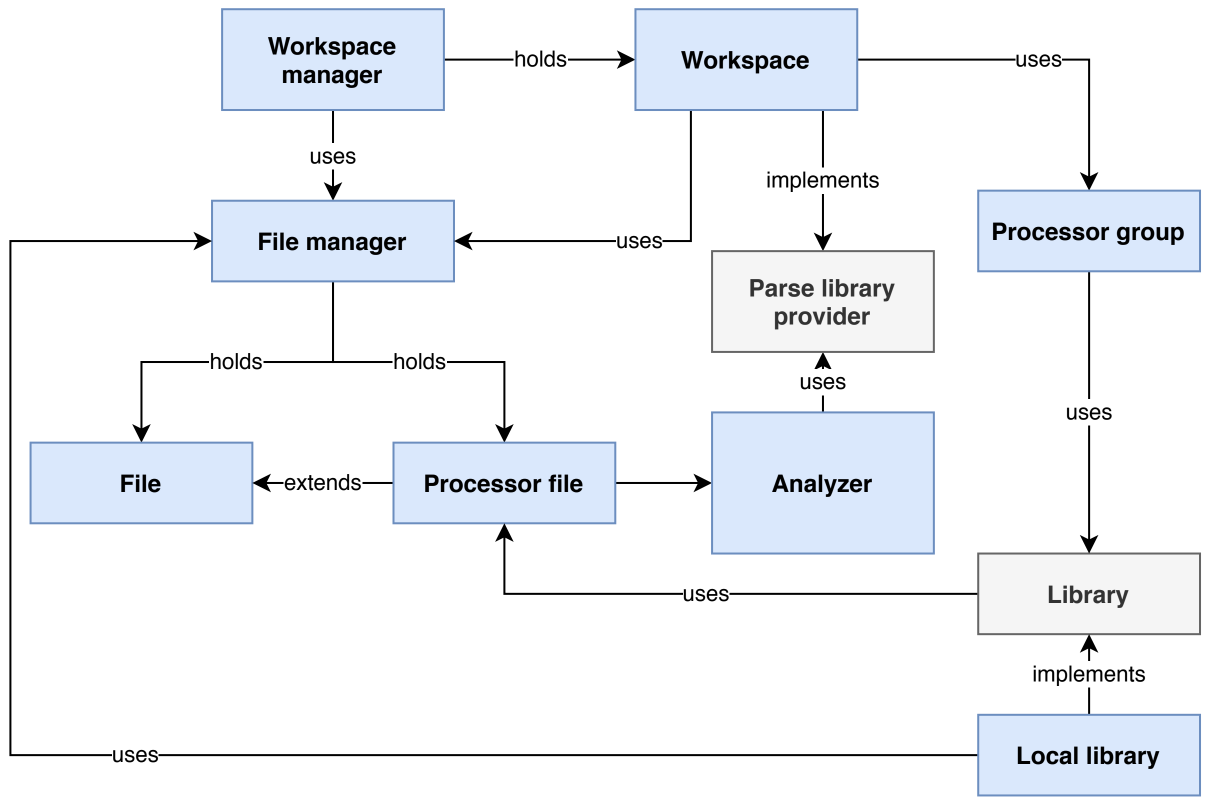

Figure 5.1: Architecture of workspace manager.

3. The method reparses the files in this list and creates new dependants, based on the dependencies reported from the parsing.

4. It checks for the files that are no longer in use (former dependencies) and closes them.

The workspace also ensures the correct closure of the file via `didClose` method. It works as follows:

- If the closed file is a dependency of some other file, it cannot be removed completely from the file manager, as it is still in use. The file manager is rather notified that the file was closed in the editor.

- If it is not a dependency, the method checks for its dependants and closes them.

5.5 File Representation

parser_library/src/workspaces/file.h
parser_library/src/workspaces/file_impl.h
parser_library/src/workspaces/file.h
parser_library/src/workspaces/processor.h
parser_library/src/workspaces/processor_file_impl.h

The file manager maintains all files across different workspaces. It distinguishes between regular, non-HLASM files and processable, HLASM files by using different representations.

The representation of a regular file (called *file*) is capable of providing its file names, its contents and changing its state upon file-oriented LSP requests, i.e didChange, didClose and didOpen.

The representation of processor files is defined by *processor_file* class, which derives from both *file* and *processor* abstract classes. The *processor* is an interface which is capable of actual processing (parsing). Its only implementation is processor file.

When the `parse` method is invoked, the processor file initializes new analyzer, uses it for the parsing and rebuilds its dependencies list, closing the unwanted ones. When the parsing is finished, it keeps the instance of the analyzer and provides its parsing results when requested.

5.6 Library path resolution

parser_library/src/workspaces/-
parse_lib_provider.h
parser_library/src/workspaces/library.h

The libraries are resolved using a *parse_lib_provider* interface. Whenever a component is to be used for dependency handling, it implements this interface and may be used by analyzer for this purpose.

The parse lib provider's `parse_library` method is passed the name of the library, the current context tables and the library data, which state what kind of processing is the analyzer using. When `parse_library` returns, library (i.e. a macro or COPY file) with the specified name is parsed and properly added to the context.

The workspace is the most important implementation of the `parse_library_provider` interface. It provides libraries based on the processor groups configuration described in section 5.2.

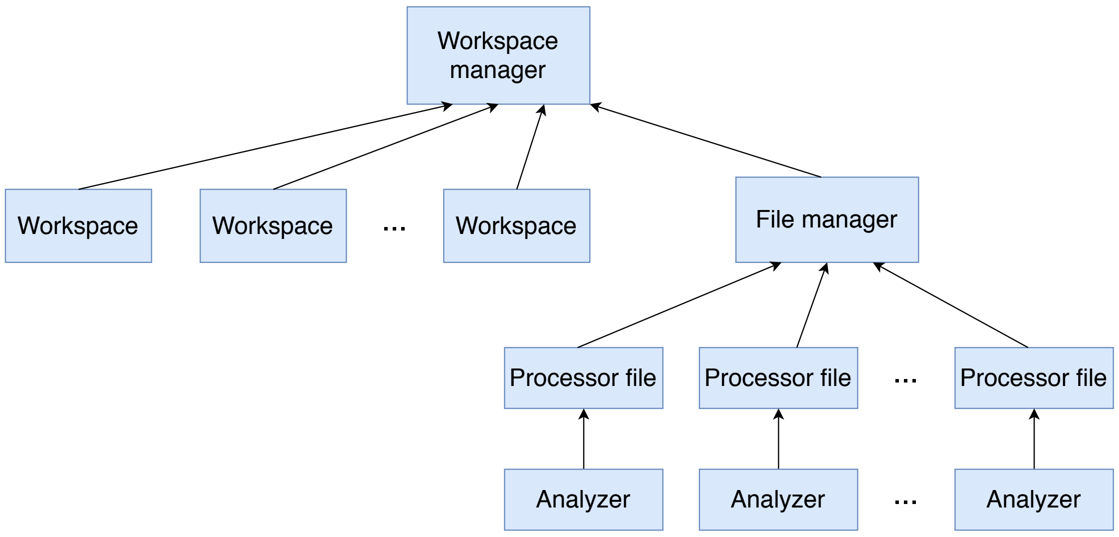

Figure 5.2: Hierarchy of diagnostics collection in the workspace manager component

5.7 Diagnostics

```
parser_library/src/diagnosable.h
parser_library/src/diagnosable_impl.h
parser_library/src/diagnostic.h
```

A diagnostic is used to indicate a problem with source files, such as a compiler error or a warning. Some diagnostics are created in almost every component of the parser when it finds a problem with a source code. Diagnostics are also used in workspace to indicate problems with configuration files. After each parsing, we need to collect all the diagnostics from all the instances of all the components and pass them to the language server.

The components capable of collecting the diagnostics are organized in a tree where the root is the workspace manager. Starting from the root, each component collects the diagnostics of those children that are again capable of collecting or generating diagnostics.

To enforce this behavior, all of these components implement the *diagnosable* interface. Its functionality is simple, it is used to add diagnostics, show his own and collect them from other diagnosable members. Each component that implements the interface is required to collect diagnostics from diagnosable objects it owns. In the result, one call of `collect_diags` from the root of the tree collects all diagnostics that were created since last such call.

The diagnosable hierarchy of workspace manager component is shown in fig. 5.2.

6. Analyzer

The role of the analyzer is to provide a facade over objects and methods to create a simple interface for lexical and semantic processing (analyzing) of a single HLASM source file. The output of the analysis is the basic input of the LSP server.

6.1 Overview

`parser_library/src/analyzer.h`

The analyzer is composed of several sub-components, all required to properly process the file.

LSP data collector collects and retrieves all LSP information created while processing the file.

HLASM context tables hold information about the context of the processed HLASM source code.

Lexer–Parser sub-components simplify the processing interface and ease the use of this component. They are needed to create a source file parser.

Processing manager executes the main loop where the file is processed.

LSP data collector is required by Lexer-Parser sub-components. They are composed into the parser object required by processing manager. HLASM context tables are used by the manager and the sub-components as well.

The components together contribute to the proper functionality of the method `analyze`. It processes a provided file and fills LSP data collector from which LSP information can be further retrieved.

6.1.1 Construction

In order to parse a HLASM file, the analyzer class is constructed with the following parameters:

- *Name and content of a file.*
- *Parse library provider* – an object responsible for resolving source file dependencies. The dependencies are only discovered during the analysis, so it is not possible to provide the files beforehand.
- *Processing tracer* (see chapter 7).

When this constructor is used, the analyzer creates HLASM context tables and processes the provided source as an open-code. We say that the analyzer has *owner semantics*; it is the owner of the context tables.

The analyzer provides *reference semantics* as well (holding just a reference of the context tables). The provided source is not treated as an open-code, rather as an external file dependency. The constructor of an analyzer with reference semantics adds the following two parameters to the previous one:

- *HLASM context tables reference* — belongs to the owning open-code analyzer.
- *Library data* — states how the dependency file should be treated (see section 6.3.4).

This constructor is called within open-code analyzer by its sub-components when they use the *Parse library provider*.

To sum up, after the analyzer is constructed, it analyzes the provided source file. As a result, it updates HLASM context tables and provides a list of diagnostics linked to the file, highlighting, list of symbol definitions, etc.

6.2 LSP data collector

parser_library/src/semantics/-
lsp_info_processor.h

The data collection is necessary to be able to reply to the LSP requests without the need to re-parse. During the parsing process, a component called *LSP info processor* processes and stores this information. The main goal of this component is to collect as much information as possible to provide meaningful and complex replies to the LSP requests while maintaining the memory and parse-time overhead negligible.

LSP info processor is invoked after each parsed and processed statement to collect and store the information it needs inside the *LSP context* (part of HLASM context).

6.2.1 Supported LSP language features

The plugin implements four LSP language features:

hover The *hover* feature is invoked whenever a user moves his mouse cursor over a symbol for a short period of time. Typically a box with the information about the selected symbol appears right next to it.

complete The *complete* feature may be triggered by a custom set of events such as typing a specific character. The server responds with a list of possible correct options that can be inserted into the particular position.

go_to_definition The *go_to_definition* feature is invoked manually from the editor by selecting a symbol and consequently invoking the `go_to_definition` command. The editor "jumps" to the location of the currently selected symbol's definition by moving the cursor to that location.

references The *references* feature is invoked in a similar manner as the *go_to_definition* feature. But the results of the *references* feature are displayed as a list of all references to the selected symbol in the project, not just the definition of it.

6.2.2 Supported HLASM symbols

The symbols, on which the user might call mentioned LSP features, are *instruction symbols*, *variable symbols*, *sequence symbols* and *ordinary symbols*.

The *references* and the *go_to_definition* features are very similar for each symbol type and in most cases work as described above.

However, there are two exceptions to the standard behavior of the *go_to_definition* feature. First, the command jumps to the definition of an instruction symbol only for macros (to the macro definition file). For the built-in instructions, the feature simply jumps to the first occurrence of the instruction. Second, the command used on an ordinary COPY symbol jumps to the corresponding copy file.

On the other hand, the responses to the *hover* and the *complete* features vary for each symbol type and are described in the following tables:

Symbol Type	Hover contents
instruction	the type of the instruction, the syntax of its parameters the version (macros only), the documentation
variable	the type of the variable — bool/string/number
sequence	the position of the definition
ordinary (COPY)	absolute/relocatable, the value, the values of attributes the name of the copy file

Table 6.1: The contents of the *hover* feature for each symbol type.

Symbol Type	Trigger Characters, Events	Response
instruction	A-Z,@,$,# after any number of spaces from the start of the line	built-in HLASM instructions + already used macros
variable	&	variable symbols defined before the current line in the current scope
sequence	.	sequence symbols defined before the current line
ordinary	*not implemented*	*not implemented*

Table 6.2: The trigger event and the reponse to the *complete* feature for each symbol type.

6.3 Processing manager

`parser_library/src/processing/processing_manager.h`

Processing manager is a major component in the processing of a HLASM source file. It decides which stream of statements is to be processed and how statements are going to be processed. It contains components responsible for instruction interpretation as well as instruction format validation.

6.3.1 Overview

Following objects passed by analyzer serve as an input for the processing manager:

- *Parser* that provides statements from the processed file. Further on in this chapter, we will refer to the parser as to the *Opencode statement provider*.
- *HLASM context tables* that hold current state of the parsed code.
- *Library data* defining the initial state of the manager (whether the file is copy member, macro definition, etc.; see section 6.3.4).
- *Name* of the processed file.
- *Parse library provider* to solve source file dependencies.
- *Statement fields parser* for parsing yet unresolved statement fields.
- *Processing tracer* for tracing processed statements (see chapter 7).

6.3.2 Composition

As the processing of the HLASM source file is rather complicated, we define a complex set of abstraction objects over the complicated assembling of HLASM language:

Statement provider (see section 6.3.6) is able to produce `statement` structures. Its functionality is to provide statements from its various statement sources (e.g., a source file for Opencode provider, a macro/copy definition for Macro/Copy provider).

Statement processor (see section 6.3.5) is an object that takes statement structures from a provider. Then, it performs a specific action with the acquired statement; namely, stores it into macro/copy definition (*Macro/Copy processor*) or looks for sequence symbol (*Lookahed processor*) or performs contained instruction (*Opencode processor*).

Instruction processors (see section 6.3.8) help opencode statement processor in performing actions with the instructions contained in a statement. Each one of four instruction processors (Macro, Assembler, Machine and Conditional Assembly IP) processes separate sub-set of a broad set of HLASM instructions.

Instruction format validators (see section 6.4) are used by instruction processors. They are provided evaluated operands of an instruction and serve to validate their correctness.

Processing manager encapsulates above mentioned objects and determines which processor/provider will be used next (see fig. 6.1).

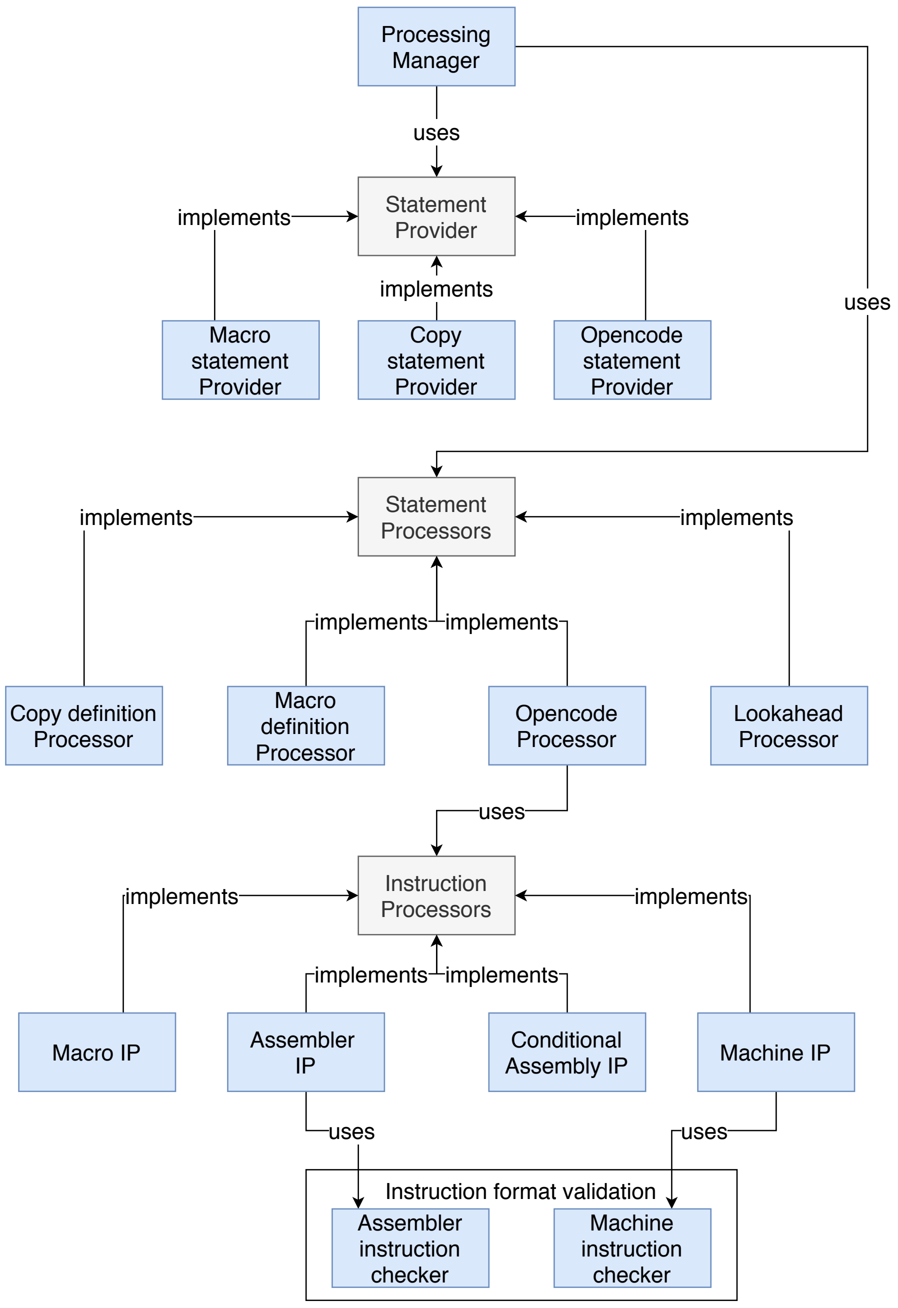

Figure 6.1: The architecture of Processing manager

6.3.3 The main loop of manager

Processing manager contains an array of active statement processors and an array of active statement providers. It is in the control of which processor–provider pair currently operates.

The main processing loop works with the currently operating processor and provider. In the loop body, statement provider provides next statement for statement processor that processes it accordingly. The loop breaks when all processors finish work and none of them is active.

When provider ends its statement stream or processor finishes its work, it is replaced with another. The following rules apply:

1. When a processor finishes its work, the next processor is selected from the array.
2. When a provider finishes — before the next provider is selected from the array — manager checks whether it triggers the termination of the current processor as well (see *terminal condition* in section 6.3.5). If true, perform rule 1.

6.3.4 Initial state of manager

It is important which statement provider and processor is assigned to the manager as a default. The manager determines this from *library data* passed by analyzer.

Library data contain a file name and an enumeration indicating a kind of the file that is being parsed — *processing kind*.

Ordinary processing kind states that the file being processed is the main source file (in HLASM called open-code). It is the first file to be processed. With this information, manager initializes all statement providers and *only* opencode processor. This initial state is applied when analyzer has owner semantics.

Copy and *Macro* processing kinds state that manager will process source code that contains copy or macro definition respectively. Hence, *only* copy definition processor or macro definition processor is initialized. Also, all statement providers but the macro statement provider are initialized as no macros will be visited nor needed as a statement source when processing new source code. The library data is passed when analyzer has reference semantics.

6.3.5 Statement Processors

`parser_library/src/processing/-statement_processors/statement_processor.h`

The motivation in distinguishing different statement processors was the complexity of HLASM language. There are many cases when the same statements require different processing under different circumstances (e.g. COPY instruction in macro is handled differently than in opencode, or lookahead mode can accept statements that would fail when processed by ordinary processing).

During processing, statement processing kinds can be nested. Hence, statement processors are dynamically assigned to the manager when needed and removed from it when they finish. This happens when the processor encounters specific statement (e.g. statement with a special instruction or non previously defined sequence symbol, see table 6.3). For this purpose, they use *processing state listener* interface (implemented by processing manager) that tells the manager to change the current processor.

Listing 10 An example of deferred statement in code.

```
*VALUE OF INSTRUCTION IN DEFERRED STATEMENT IS PARAMETER OF MACRO MAC
         MACRO
         MAC      &INSTR
         &INSTR   3(2,0),13       ← deferred statement
         MEND
```

6.3.5.1 Statement structure

`parser_library/src/context/hlasm_statement.h`
`parser_library/src/processing/statement.h`
`parser_library/src/semantics/statement.h`

Statement consists of *statement fields* — *label field*, *instruction field*, *operands field*, *remark field*. It is used by statement processors and produced by statement providers.

The abstract class *HLASM statement* is the ancestor for all statement related classes. Then, there are abstract classes *deferred statement* and *resolved statement*. Deferred statement has its operand field stored in uresolved — deferred — format (in code stored as string). This statement is created when actual instruction is not yet known prior to the statement creation (see listing 10). Resolved statements are complementary to the deferred statements as their instruction — as well as operand format — is known prior to the statement creation.

6.3.5.2 Copy and Macro definition Processors

`parser_library/src/processing/-`
`statement_processors/copy_processor.h`
`parser_library/src/processing/-`
`statement_processors/macrodef_processor.h`

Both of these statement processors handle statement collecting, forming definition structure and storing it into HLASM context tables. They come into effect when COPY instruction or macro definition is encountered in the source code.

The statements collected inside copy or macro definitions are mainly deferred statements. That is because variable symbols can not be resolved inside the definition and because HLASM allows instruction aliasing (renaming instructions). Therefore, during the processing of a definition, as the instruction field is parsed, the format of its operands is unknown. It is fully deduced when the definition is handed over to the provider and processed by the opencode processor.

However, some statements in the macro and copy definitions forbid aliasing and the operand format can be deduced immediately (e.g. conditional assembly instructions in macro definition). This leads to the processors necessity to ask the provider to retrieve the statement with correct format – accordingly to the deduced one based on the instruction being provided (see section 6.3.6.1).

6.3.5.3 Lookahead Processor

`parser_library/src/processing/-`
`statement_processors/lookahead_processor.h`

Lookahead processor is activated when currently processed conditional assembly statement requires a value of undefined ordinary or sequence symbol. It looks through the succeeding statements and finishes when the target symbol is found or when all statement providers finish. Then the processing continues from where the lookahead started.

Processor	END instruction	COPY instruction	MACRO instruction	MEND instruction	undefined symbol
Opencode	finish	start Copy	start Macro	continue	start Lookahead
Copy	continue	continue	continue	continue	continue
Macro	continue	start Copy	continue	finish	continue
Lookahead	finish	start Copy	continue	continue	continue

Table 6.3: Description of statement processor changes.

6.3.5.4 Opencode Processor

`parser_library/src/processing/-statement_processors/ordinary_processor.h`

The functionality of Opencode processor (`ordinary_processor` class) can be described as follows:

1. If a model statement is encountered (see section 2.2.3.1), it substitutes the variable symbols and resolves the statement.
2. It checks statement for validity.
3. It performs instruction by updating HLASM context tables with the help of *instruction processors*.
4. It is passed *processing tracer* by the manager. Each time a statement is processed by opencode processor, it triggers processing tracer. The tracer serves as a listener pattern used by *Macro tracer* (see chapter 7).

In the table 6.3, we can see that it does not have a field that starts Opencode processor. That is because this processor is set as a default by the manager. Further, Copy processor does not finish itself during its work as it can only be finished by its *terminal condition* (see table 6.4).

Terminal condition can be triggered by a finishing provider. It indicates that the processor needs to finish its work when a specific provider exhausted its statement stream.

6.3.6 Statement Providers

`parser_library/src/processing/-statement_providers/statement_provider.h`

Due to the macro definitions, copy file includes and statement generation, it is difficult to state which statement should be processed next. For this reason, we define abstraction over various sources of statements called *statement providers*.

In contrary to statement processors, statement providers are ordered based on the priority (lower index, greater priority):

1. Macro definition statement provider
2. Copy definition statement provider
3. Opencode statement provider

Processors	Macro provider ends	Copy provider ends	Opencode provider ends
Opencode	continue	continue	finish
Copy	continue	continue	finish
Macro	finish	continue	finish
Lookahead	finish	continue	finish

Table 6.4: Description of statement processors' terminal condition.

In each iteration of processing manager (see section 6.3.3), providers are asked whether they have statements to provide based on the ordering. That is because after each iteration, a provider with greater priority than the previously used one can be activated.

For the main loop to be correctly defined, the end of opencode provider triggers terminal condition for all statement processors. Hence, when opencode provider finishes then all the processors finish as well and the processing ends (see table 6.4).

6.3.6.1 Statement passing

In HLASM language, it is difficult to parse statements into one common structure due to its *representational ambiguity*; the major difference between operand fields of different instruction formats. Moreover, when parsing statements, the instruction format can be yet unknown. Therefore, operand fields are stored as strings. This means that during statement passing when instruction format is deduced, the provider has responsibility to produce correct statement format. The following steps are applied in the statement passing (also see fig. 6.2):

1. Provider retrieves the instruction field part of the statement.
2. Provider calls processor method `get_processing_status` with instruction field as a parameter.
3. Return value of the call determines the required format of the operand field for the processor; the whole statemement can be retrieved correctly.
4. Provider returns statement with correct format to the processor.

6.3.6.2 Copy and Macro definition Provider

`parser_library/src/processing/-statement_providers/-copy_statement_provider.h`
`parser_library/src/processing/-statement_providers/-macro_statement_provider.h`

These providers are activated when COPY instruction copies a file into the source code or when a macro is visited, respectively. They provide a sequence of statements to an arbitrary processor until all statements from the copy or macro definition are provided. After that, if there is no nested invocation, a provider with lower priority is selected.

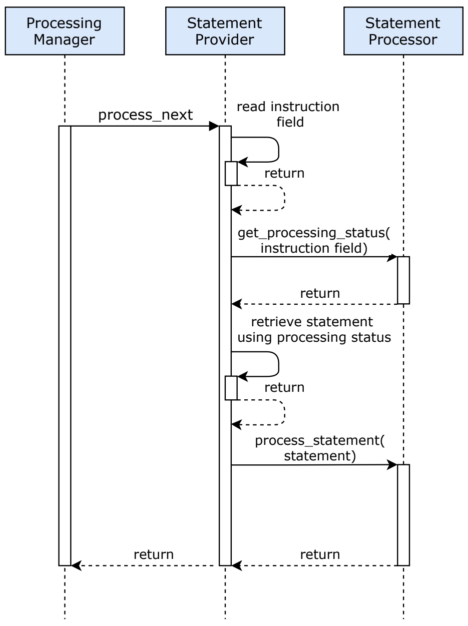

Figure 6.2: The process of statement passing.

6.3.6.3 Opencode Provider

`parser_library/src/processing/-opencode_provider.h`
`parser_library/src/parsing/parser_impl.h`

Opencode provider is active as long as there are statements in the source file. It retrieves statements from the source code with help of lexer and parser (see section 6.6).

6.3.7 Statement field parser

`parser_library/src/processing/-statement_fields_parser.h`

Statement field parser is an interface passed to the statement providers by processing manager. It is implemented by the parser (see section 6.6).

At first, it is used during statement passing. In some cases provider is requested a specific format of a string-stored statement. The string is re-parsed with the according format. Then the field is returned back to the statement provider.

Another use of field parser is in opencode processor as model statements are resolved there. After variable symbol substitution, the resulting string field is re-parsed with field parser.

6.3.8 Instruction processors

`parser_library/src/processing/-instruction_sets`

Opencode processor divides processing of HLASM instruction types into several *instruction processors*. Each processor is responsible for processing instructions that belong to one instruction type.

IP	Processed instructions
Assembler	*SECT, COM, LOCTR, EQU, DC, DS, COPY, EXTRN, ORG
Machine	*Instruction format validation only*
Macro	*ANY*
Conditional Assembly	SET*, GBL*, ANOP, ACTR, AGO, AIF, MACRO, MEND

Table 6.5: Table of instructions that are processed by instruction processors.

As a format of some instruction kinds can be rather complicated, instruction processors contain *Instruction format validators* (see section 6.4). They check the statement to validate the correctness of used operand format as well as the correctness of the actual operand values.

During the instruction processing, processors work with HLASM *expressions* (see section 6.3.8.1, section 6.3.8.3). They need to be evaluated to correctly perform the processing.

There are four specialized instruction processors:

Macro IP looks up for macro definition in HLASM context tables and calls it.

Assembler and Machine IP processes assembler and machine instructions (see section 2.2.1.3 section 2.2.1.2) to retain consistency in HLASM context tables.

Conditional assembly IP executes conditional assembly instructions (see section 2.2.3.2).

See the current list of processed instruction in table 6.5.

HLASM differentiates two kinds of expressions: *Conditional Assembly* (CA) and *Machine* expressions. CA expressions appear in conditional assembly, which is processed during compilation. Machine expressions are used with assembler and machine instructions.

6.3.8.1 CA Expressions

```
parser_library/src/expressions/expression.h
parser_library/src/expressions/arithmetic_expression.h
parser_library/src/expressions/character_expression.h
parser_library/src/expressions/keyword_expression.h
parser_library/src/expressions/logic_expression.h
parser_library/src/expressions/numeric_wrapper.h
```

HLASM evaluates CA expressions during assembly generation. For further details, refer to the section 2.2.3.2.

We employ the ANTLR 4 Parse-Tree Visitors during the expression evaluation. For further detail on ANTLR, refer to section 9.1.

In this section, we address the representation and functionality of the expressions themselves. Coupling the expressions with grammar and their evaluation in context is further discussed in section 6.3.8.2.

HLASM CA expression is conceptually similar to the expressions in other languages: they support unary and binary operators, functions, variables and literals. Evaluation of expressions is further reviewed in section 6.3.8.2. In HLASM, each expression has a type.

- *Arithmetic*,

- *Logic*,
- *Character*

expressions are supported. We implement the logic in the following classes:

`expression` A pure virtual class that defines a shared interface, operators, and functions. The class also implements evaluation logic for terms and factors.

`diagnostic_op` The concept of *diagnostics* is fundamental. During the evaluation of an expression, an error can occur (syntactic or semantic). Hence, we try to improve the user experience by reporting diagnostics. Each instance of `expression` has a pointer to `diagnostic_op` associated to it. If the pointer is `null`, it is considered error-free. During the evaluation of a child expression, the parent checks for errors and propagates the error upwards. Checking and propagating of an error is implemented by `copy_return_on_error` macro, which one should call immediately before the creation of a new expression during evaluation.

The `expression` class implements the evaluation as follows: A `std::deque` of `expression` pointers is passed. The evaluation iterates the list from left to right. Functions, binary, and unary operators consume the rest of the deque.

Some expression symbols can be either HLASM keywords or variable identifiers (see example in listing 11). Therefore, the resolution of symbols is complicated and cannot be done straight, but instead during the evaluation-time. The order of the expression's terms and the previous evaluation context is crucial for the disambiguation.

`keyword_expression` Helper class that represents HLASM keywords in expressions. It determines a keyword type from a string, containing its arity (unary, binary) and priority.

`logic_expression` Represents a boolean expression.

`arithmetic_expression` Represents an arithmetic expression.

`arithmetic_logic_expr_wrapper` HLASM language supports expressions with operands of mixed types. For more straightforward and readable use of arithmetic and logical expressions, this class wraps them under one class.

`character_expression` Represents a character expression.

`ebcdic_encoding` This class defines a custom EBCDIC literal and provides helper functions for conversion between EBCDIC and ASCII. EBCDIC is a character encoding used in the IBM mainframe. It has a different layout than ASCII.

`error_messages` It is a static class with list of all `diagnostic_op` that can be generated from expressions.

Listing 11 An example of using keywords (AND and NOT) as variable names resulting in a confusingly valid expression (line 3).

```
name        operation   operands

AND         EQU         1
NOT         EQU         0
            AIF         (NOT AND AND AND).LAB   < EVALUATES TO (!(1 & 1))
```

6.3.8.2 CA expression evaluation

parser_library/src/expressions/visitors/-
expression_analyzer.h
parser_library/src/expressions/visitors/-
expression_evaluator.h
parser_library/src/expressions/-
evaluation_context.h

In the previous section, we described the representation of the CA expressions themselves. In this section, we explain the coupling of CA expressions with grammar via visitor.

The `expression_evaluator` encapsulates the coupling logic between the grammar and the expression logic. That is, the evaluator has a notion about grammar, which translates into C++ expression logic.

The top-level expression first gathers a list of space-separated expressions. The evaluation must be done using a list from left to right (not using a tree) as any token may be a keyword (such as AND operator) or variable identifier, depending on a position in an expression (using language keywords as identifiers is allowed in HLASM). For a better understanding, see listing 11. `expression::evaluate` provides the disambiguation (see section 6.3.8.1).

During its work, evaluator substitutes variable and ordinary symbols for their values. To know which values to substitute, evaluator is given *evaluation context*. It consists of objects that are required for correct evaluation: *HLASM context* for symbol values, *attribute provider* for values of symbol attributes that are not yet defined and *library provider* for evaluation of some types of symbol attributes as well.

Lookahead (see section 6.3.5.3) is triggered in conditional assembly expressions when evaluation visits yet undefined ordinary symbol. As this can be rather demanding operation, expression evaluator uses *expression analyzer*. It looks for all the undefined symbol references in expression and collects them to a common collection. Then, the lookahead is triggered to look for all references in the collection. Hence, it is triggered once per expression rather than any time an undefined symbol reference is found.

6.3.8.3 Machine expressions

parser_library/src/expressions/mach_expression.h
parser_library/src/expressions/mach_expr_term.h
parser_library/src/expressions/mach_operator.h

In HLASM, machine expressions are used as operands of machine and assembler instructions. Their result may be a simple absolute number or an address.

We use a standard infix tree representation of expressions. There is an interface `machine_expression` which is implemented by several classes that represent operators and terms. Each binary operator holds two expressions — left and right operands. Terms

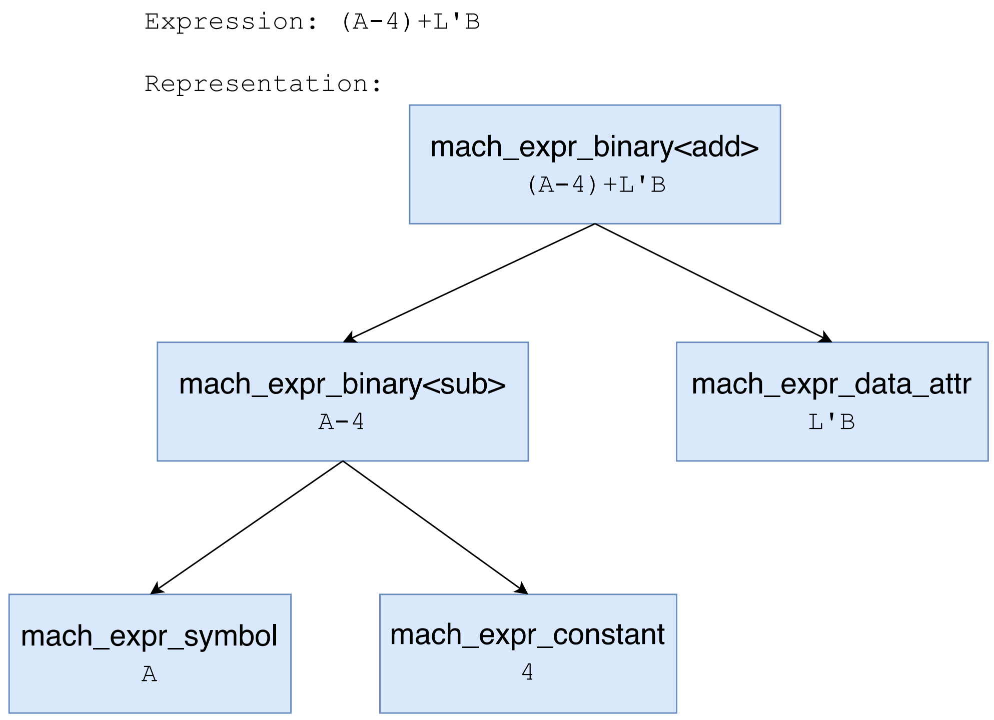

Figure 6.3: Example representation of the machine expression `(A-4)+L'B`.

are leaf classes that do not hold any other expressions and directly represent a value. There are several classes representing different terms valid in machine expressions:

- `mach_expr_constant` represents simply a number.
- `mach_expr_symbol` represents an ordinary symbol.
- `mach_expr_data_attr` represents attribute of a symbol (e.g. `L'SYM` is length of symbol `SYM`)
- `mach_expr_location_counter` represents location counter represented by asterisk in expressions.
- `mach_expr_self_def` represents self defining term (e.g. `X'1F'`)

Figure 6.3 shows an example representation for one concrete expression.

Machine expressions are also able to evaluate the expressions they represent. The evaluation is done in a recursive manner. It is fairly simple when there are no symbols used in the expression — each node in the tree simply computes the result with basic arithmetic operations.

However, the process can get tricky since expressions may contain e.g. `mach_expr_symbol` whose value is dependant on symbols defined in other parts of source code. Moreover, result of a machine expression may be an absolute value (a number) or relocatable value (an address). The process of symbols resolution is explained in section 6.7.8.4.

6.4 Instruction format validation

parser_library/src/context/instruction.h
parser_library/src/checking/operand.h
parser_library/src/checking/instr_operand.h

One of the essential ways to provide results of the parsing to the user is through error messages. Many of these messages are created in *Instruction checker* which validates the usage of different kinds of instructions.

Instruction checker is an abstract class for various types of instructions. Its `check` method is being called from the instruction processors 6.3 to check whether the specific instruction is used with correct parameters. As assembler and machine instructions have different formats, we derive separate *assembler* and *machine* checkers from the instruction checker. CA instructions do not have a derived checker class as they are all being checked during their interpretation.

The checkers need an access to the definitions of all possible instructions. These instructions are stored statically inside an object called *instruction*. It consists of 4 different containers:

- *machine_instructions* is a map of instruction names to machine instruction object, which contains various data such as format, size or vector of instruction's operands.
- *mnemonic_codes* maps instruction names to their mnemonic code. The mnemonic codes are simplified versions of specific machine instructions, substituting one of the operands by a default value. The mnemonic code objects provides a list of operands to be substituted along with the original instruction name.
- *assembler_instructions* is similar to the machine instructions. However, as the assembler instructions do not have formats, these classes only state minimum/maximum number of operands for specific instruction. In section 6.4.2, we explain how the assembler instructions are validated.
- *ca_instructions* only contains a list of possible CA instructions.

Both assembler and machine checker works in a similar manner:

1. Either assembler or machine processor calls the `check` method of its respective checker. This method accepts the instruction name, the vector of used operands, the range of statement and the diagnostic collector.
2. Checker finds the correct instruction based on the provided name and calls the `check` method of its instruction class, along with the same parameters as mentioned above.
3. The instruction itself compares its possible operands with the used operands.
4. More validations may be necessary, based on the instruction.
5. In case of mismatch, a diagnostic is added to the passed diagnostic container.

6.4.1 Machine instruction checker

parser_library/src/checking/-
instruction_checker.h
parser_library/src/context/instruction.h

All machine instructions have a precisely defined format which makes the validation based on these formats straightforward. Machine instructions checker operates with machine instructions and their mnemonic codes.

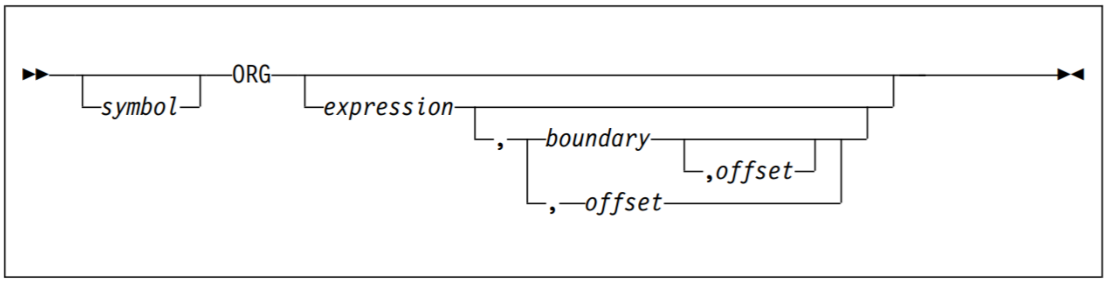

Figure 6.4: Operand diagram for the ORG instruction.

The formats are defined by several basic operands such as register or address and state which combination of these operands are acceptable. For example, instruction LR has format RR, which means it accepts only 2 arbitrary (but correct) registers.

6.4.2 Assembler instruction checker

parser_library/src/checking/asm_instr_class.h
parser_library/src/checking/asm_instr_check.h

Validation of assembler instructions is more complicated as there are no pre-defined formats for them. Each of them is described by custom operand diagrams, which demonstrate the dependencies and relations between operands of a specific instruction. An example of such diagram for the ORG instruction is shown in fig. 6.4. As an addition to the basic operands used for machine instructions, each assembler instruction might have its own operands, called keywords.

Due to these irregularities, we derive instruction-specific classes from assembler instruction class. Each of them implements the `check` method, to provide the customized checking.

6.4.2.1 Data Definition checking

parser_library/src/checking/data_definition/data_def_fields.h
parser_library/src/checking/data_definition/-
data_def_type_base.h
parser_library/src/checking/data_definition/data_def_types.h
parser_library/src/checking/data_definition/-
data_definition_operand.h

Data definition is a type of operand in HLASM. It represents data that is assembled directly into object code (see section 2.2.1.3).

Since there are many types of data definition, there is a data definition subcomponent of instruction validation. Whenever any component of the project needs information about a data definition operand, it can use this subcomponent. It analyzes each type of data definition and is able to return its length, attributes and check its validity.

Each type is different and many have special conditions that must be met to be valid. That is why there is an abstract class `data_def_type_base`, which has 38 implementations — one for each type (including type extensions). The types are then available in a static associative map that maps names of types to their representations.

6.5 Lexer

parser_library/src/lexing/lexer.h

Lexer's responsibility 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 a location in the source text,
- has the ability to check whether all characters are valid in the HLASM source,
- can 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.

As previously mentioned, we designed a custom lexer for HLASM. We had a number of reasons to do so. HLASM language is complex. It was first introduced several decades ago and, during this long time, the language was subjected to development. Such a long time period has made the HLASM language complex. Also, it contains some aggressive features, for example, `AREAD` or `COPY`, that can alter the source code at parse time.

Conventional lexing tools are most often based on regular expressions. As discussed above, there are several difficulties that one must consider while designing lexer for this particular language. A regular expression-based lexer would be too difficult or even impossible to design[1].

6.5.1 Source file encodings

Source code encodings differ for the used libraries. All strings are encoded in `UTF` as follows:

`UTF-8` LSP string encoding,

`UTF-16` offsets (positions in source code) in LSP,

`UTF-32` ANTLR 4 source code representation.

6.5.2 Lexer components

parser_library/src/lexing/input_source.h
parser_library/src/lexing/token.h
parser_library/src/lexing/token_factory.h

Beside of the custom lexer, we altered ANTLR's classes `Token`, `TokenFactory` and `ANTLRInputStream` (see section 9.1). The reason was to add custom attributes to token that are vital for later stages of the HLASM code analysis (parsing, semantic analyses, etc.). Lexer functionality is implemented in following classes (see fig. 6.5):

`token` implements ANTLR's class `Token` and extends it by adding properties important for location of the token within the input stream. As the LSP protocol works with offsets encoded in `UTF-16` and ANTLR 4 works with `UTF-32` encoding, we add attributes for `UTF-16` positions too.

Token does not carry the actual text from the source but instead references the position in code (unlike `CommonToken`). Note that the position of a token is vital for further analysis.

[1]One could match separate characters from the input and let the parser or semantic analysis deal with some of the described problems. This drastic solution would cost performance, as parsers are usually more performance demanding.

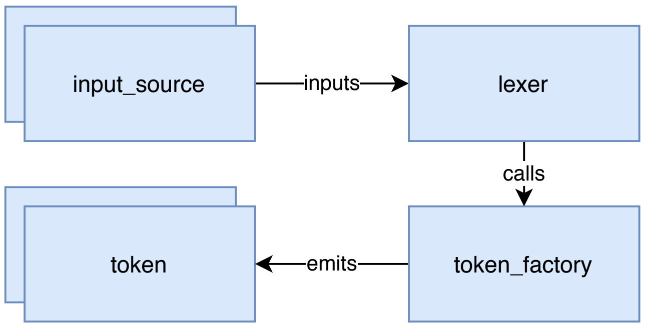

Figure 6.5: Lexer architecture overview. Note, there are two `input_sources` and there are many `tokens` generated.

IGNORED	sequence of characters ignored in processing
COMMENT	commentary statements
EOLLN	token signalling the end of statement
SPACE	a sequence of spaces
IDENTIFIER	symbol identifier
ORDSYMBOL	Ordinary symbol identifier
PROCESS	process statement token
NUM	number
ATTR	apostrophe that serves as attribute reference
ASTERISK, SLASH, MINUS, PLUS, LT, GT, EQUALS, LPAR, RPAR	expression tokens
DOT, COMMA, APOSTROPHE, AMPERSAND, VERTICAL	special meaning tokens

Table 6.6: Enumeration of tokens.

Interesting remark of HLASM language complexity is absence of *string* token (see table 6.6). Lexer does not generate this token due to the existence of model statements (see section 2.2.3.1). There, variable symbol can be written anywhere in the statement (even in the middle of the string), what significantly restricts lexer.

`token_factory` produces tokens of previously described custom type `token`.

`input_source` implements `ANTLRInputStream` which encapsulates source code. This implementation adds API for resetting, rewinding and rewriting input.

Beware of the usage of `UTF` encodings: `_data` (source code string) and positions/indices in API are in `UTF-32`; `getText` returns `UTF-8` string.

`lexer` is based on ANTLR's `TokenSource` class. As most lexers, it is also, in principle, a finite state machine. The most important difference compared to conventional FSMs and other lexers is added communication interface that connects the parser and the instruction interpreter with the lexer. Unusual is also input rewinding (to support `AREAD`, for example), lexing from parallel sources (`AINSERT` buffer) and some helper API for subsequent processing stages.

Important functions:

`nextToken()` implements main functionality: lexes and emits tokens. Before lexing, the function uses the right input stream (either the source code or `AINSERT` buffer if not empty). After choosing the right input source, the lexer emits token. HLASM introduces *continuation* symbol (an arbitrary non-blank symbol in the continuation column) that breaks one logical line into two or more source-code lines. The end of one logical line indicates `EOLLN` token. Such token is important for further (syntactic and semantic) analysis.

`create_token()` creates token of given type. The lexer's internal state gives the position of the token.

`consume()` consumes character from the input stream and updates lexer's internal state (used in `create_token()`).

`lex_tokens()` lexing of most of the token types.

`lex_begin()` up to certain column, the input can be ignored (can be set in HLASM).

`lex_end()` lexes everything after continuation symbol.

6.6 Parser

Parser component takes tokens produced by lexer from token stream and recognizes HLASM statements. The parser inherits from the HLASM recognizer generated by ANTLR (see section 9.1) to provide further operations.

6.6.1 Parser workflow

`parser_library/src/parsing/parser_impl.h`
`parser_library/src/semantics/range_provider.h`

Parser (in code referenced as `parser_impl`) implements opencode statement provider interface. This means that, according to the statement passing in section 6.3.6.1, parser needs to parse each statement in *two steps*:

1. Parser calls rule `label_instr`. It parses label and instruction fields into respective structures. The operand and remark field is stored as a string.
2. After retrieving the processing format, the parser selects corresponding rule to parse operands. With the rule, it parses remaining string from the previous step.

For the means of parsing remaining strings, parser subcomponent contains actually *two parsers*. The first one parses statement after statement from a source file. The second parses the operands from the string passed by the first parser.

To achieve operands having correctly set ranges prior to the source file rather than to the passed string, the parser uses *Range provider*. It helps the second parser to have ranges of reparsed operands consistent with the ranges of other fields. It is initialized with the begin location of operand field in the statement and all ranges furtherly created in parsing are adjusted to have correct boundaries.

6.6.2 Statement structure

`parser_library/src/semantics/collector.h`
`parser_library/src/semantics/statement.h`
`parser_library/src/semantics/-`
`statement_fields.h`

During parsing of a statement, several structures are created and collected. They are `label_si`, `instruction_si`, `operand_si`, `remark_si` (*si* as semantic information). They are collected with `collector` and built into `statement_si` structure.

Label and instruction structures can contain either identifier of a symbol or — when in model statement — concatenation of strings and variable symbols. Remark field is simply just a string as it serves as a commentary statement field. Operand field contains list of operands used in the statement. They can be of several formats.

6.6.2.1 Operand formats

`parser_library/src/semantics/operand.h`

The statement processor can request parser to retrieve statements with this operand formats:

- *machine/assembler/conditional assembly/macro* – instruction operands. Each type of instruction has it's specific format.
- *model* – operands for model statements. It is a chain of strings and variable symbols.
- *deferred* – operands with not yet known format. Stored as a string.

Each operand format has corresponding *operand structure*. They all inherit abstract `operand` and each have various children for different kinds of the operand format (see fig. 6.6). Assembler and Machine operand structures inherit from *Evaluable operand*. It is a common structure for operand objects that are composed of resolvable objects (see section 6.7.8.4).

6.6.2.2 Concatenation structures

`parser_library/src/semantics/concatenation.h`

A model statement is a statement that contains a variable symbol in any of the statement fields. This variable symbol is further to be substituted by an arbitrary string and then re-parsed. Hence, the field is represented by a concatenation of helper structures. The concatenation can be further evaluated to produce the final string.

The helper structures are:

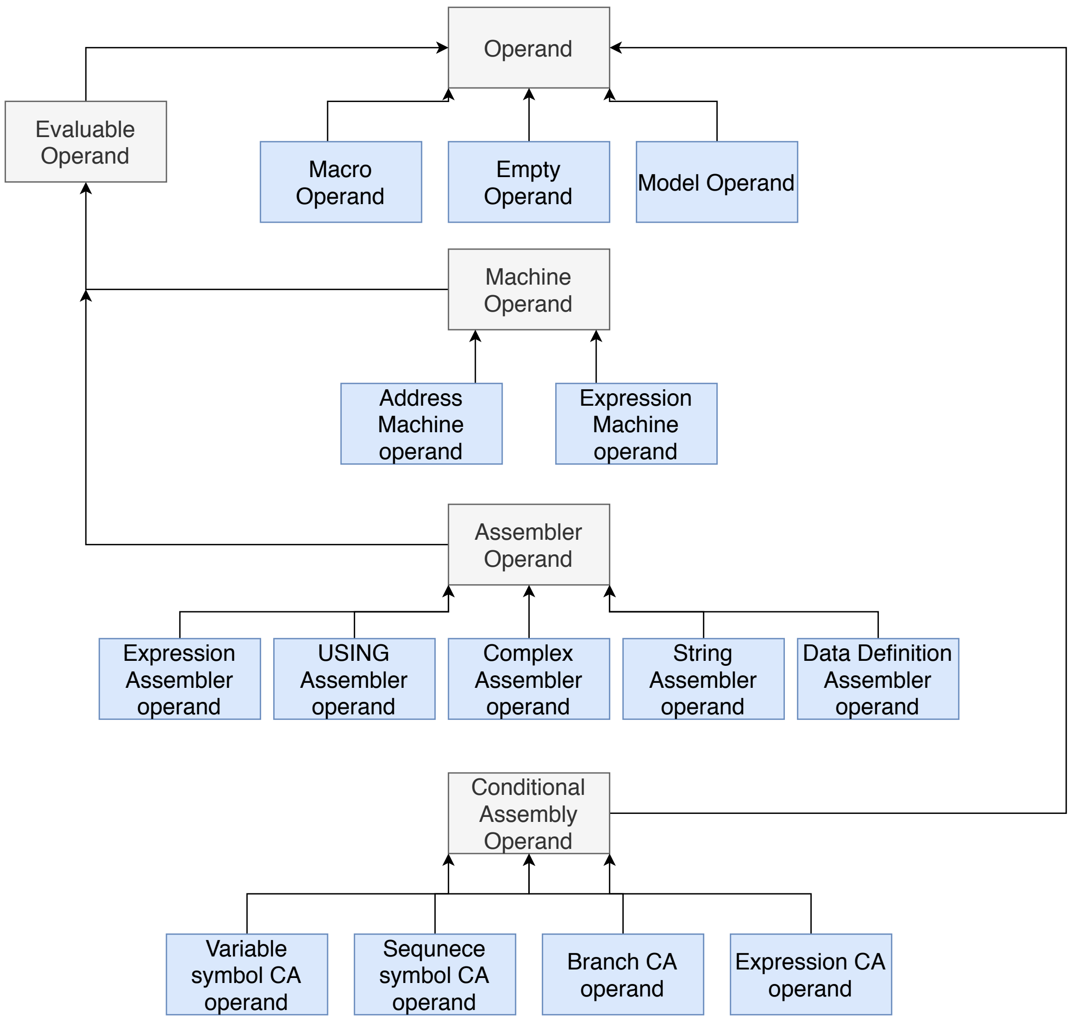

Figure 6.6: Operand structure inheritance.

- `char_str` – character string.
- `var_sym` – substitutable variable symbol.
- `dot`, `equals` – characters with special meaning.
- `sublist` – by parenthesis enclosed recursive concatenation.

6.6.3 Grammar implementation `parser_library/src/parsing/grammar`

Grammar rules describing parser are separated into several files (see the figure of all grammar rules in appendix B):

- `hlasm_parser.g4` – Top level rules are stored here.
- `lookahead_rules.g4` – Rules for lookahead mode.
- `label_field_rules.g4` – Rules taking care of label field of statement.
- `instruction_field_rules.g4` – Rules taking care of instruction field of statement.
- `operand_field_rules.g4` – Rules taking care of operand field of statement.
- `macro/machine/assembler/ca/model/deferred_operand_rules.g4` – Various operand field rules.
- `ca/asm_expression_rules.g4` – Rules for expressions.
- `data_def_rules.g4` – Rules for data definition.

6.7 HLASM context tables

parser_library/src/context/hlasm_context.h

HLASM context tables (in code referred simply as `hlasm_context`) is composition of tables and stacks that describe the state of the currently processed open-code. This structure is persistent between source files within an open-code. It is created in an analyzer and has the same lifespan.

It is composed of:

- *Macro & Copy storage* – stores macro and copy definition definitions.
- *ID storage* – stores symbol identifiers.
- *Scope stack* – stores nested macro invocations and local variable symbols.
- *Global variable symbol storage* – stores global variable symbols.
- *Source stack* – stores nested source files.
- *Processing stack* – stores stack of processings in a source file.
- *LSP context* – stores structures for LSP requests.
- *Ordinary assembly context* – encapsulates structures describing Ordinary assembly.

6.7.1 Macro definition

parser_library/src/context/macro.h

HLASM context stores visited macro definitions in the *macro strorage*.

Macro definition is represented by:

- *Macro identifier*. It identifies the macro.
- *Calling parameters*. They are assigned real value when the macro is called.
- *Block of statement*. It represents the body of the macro.
- *Block of copy nestings*. It is an array with one-to-one relation with block of statements. Each entry is a list of in-file locations that represents how much is the statement nested in COPY calls.
- *Label storage*. The storage of sequence symbols that occur in the macro definition.

When macro is called, *macro invocation* object is created. It shares the content of a respective macro definition with an exception of calling parameters as they are assigned real value passed with the call. Also, it contains index to the top statement of the invocation.

The macro invocation is stored in the context's *scope stack*.

6.7.2 Scope stack

`parser_library/src/context/code_scope.h`

This stack (stack of `code_scope` objects) holds information about the scope of variable symbols (see section 6.7.6). The scope changes when macro is visited. The initial scope is the open-code.

The stack elements contain:

- In-scope variable symbols.
- In-scope sequence symbols.
- Pointer to the macro invocation (NULL if in open-code).
- Branch counter (for ACTR instruction).

6.7.3 COPY

`parser_library/src/context/copy_member.h`

HLASM context stores visited COPY members in the *copy strorage*.

COPY member definition is much more simple than the macro definition as it does not hold any more semantic information than the sequence of statements (the definition itself).

When copy is visited, copy member invocation is created and pushed in the copy stack of last entry of the *source stack*.

6.7.4 Source stack and Processing stack

`parser_library/src/context/-processing_context.h`

This stacks are responsible for the nests of opened files (source stack) and what they are opened for (processing stack). As the relation of source entry and processing entry is one-to-many, the information is stored in two arrays rather than one.

When statement processor (see section 6.3.5) is changed (e.g. macro or copy definition is processed, lookahead is needed, ...), this information is stored in the processing stack. If a new file is opened during this change then source stack is updated as well.

Source stack contains:

- *Source file identifier*
- *Copy stack* – the nest of copy calls active for the source file.
- *Processed statement location* – data that locates last processed statement in the source file.

Processing stack contains *processing kind*.

The reasoning of organizing this two stacks in such a way is:

1. Context has enough information to fully reconstruct the statement.
2. Easy retrieval of the correct copy stack for copy statement provider.

6.7.5 ID storage

`parser_library/src/context/id_storage.h`

ID storage holds the string identifiers that are used by the open-code. It stores the string and retrieves a pointer. It is guaranteed that if two different strings with the same value are passed to the storage, the resulting pointers are equal.

It simplifies work with IDs and saves space.

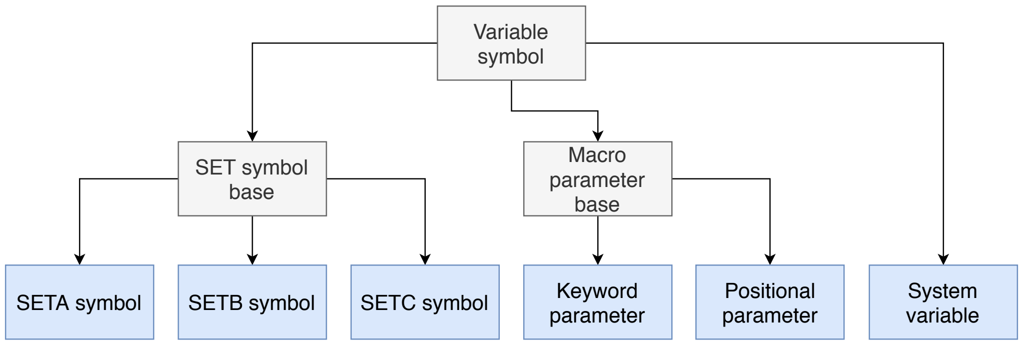

Figure 6.7: The inheritance of variable symbols.

6.7.6 Variable symbols

`parser_library/src/context/variables/variable.h`
`parser_library/src/context/variables/macro_param.h`
`parser_library/src/context/variables/set_symbol.h`
`parser_library/src/context/variables/system_variable.h`

In HLASM language, variable symbol is general term for symbols beginning with ampersand. However, they can be separated into several structures that capture a common behavior:

- *SET symbols* – represent HLASM SET symbols.
- *System variables* – represent HLASM system variables.
- *Macro parameters* – represent HLASM macro parameters.

They inherit common abstract ancestor *variable symbol*. SET symbols are further divided into *SETA*, *SETB* and *SETC* symbols. Macro parameters are divided into *keyword* and *positional* parameters (see fig. 6.7). They are stored in respective storage (global storage, scope stack, macro definition) that determines their scope.

6.7.7 LSP context

`parser_library/src/context/lsp_context.h`

The LSP context serves as the collection point for the data needed to answer the LSP requests. It is a part of the HLASM context to be able to pass on the LSP data between different parsed files.

The LSP data collector stores its values inside the LSP context tables. More about the collection of the data and their values can be found in section 6.2.

6.7.8 Ordinary assembly context

`parser_library/src/context/ordinary_assembly/-`
`ordinary_assembly_context.h`

The above described structures aimed to describe the high-level part of the language (code generation). As we move closer to the resulting object code of the source file, the

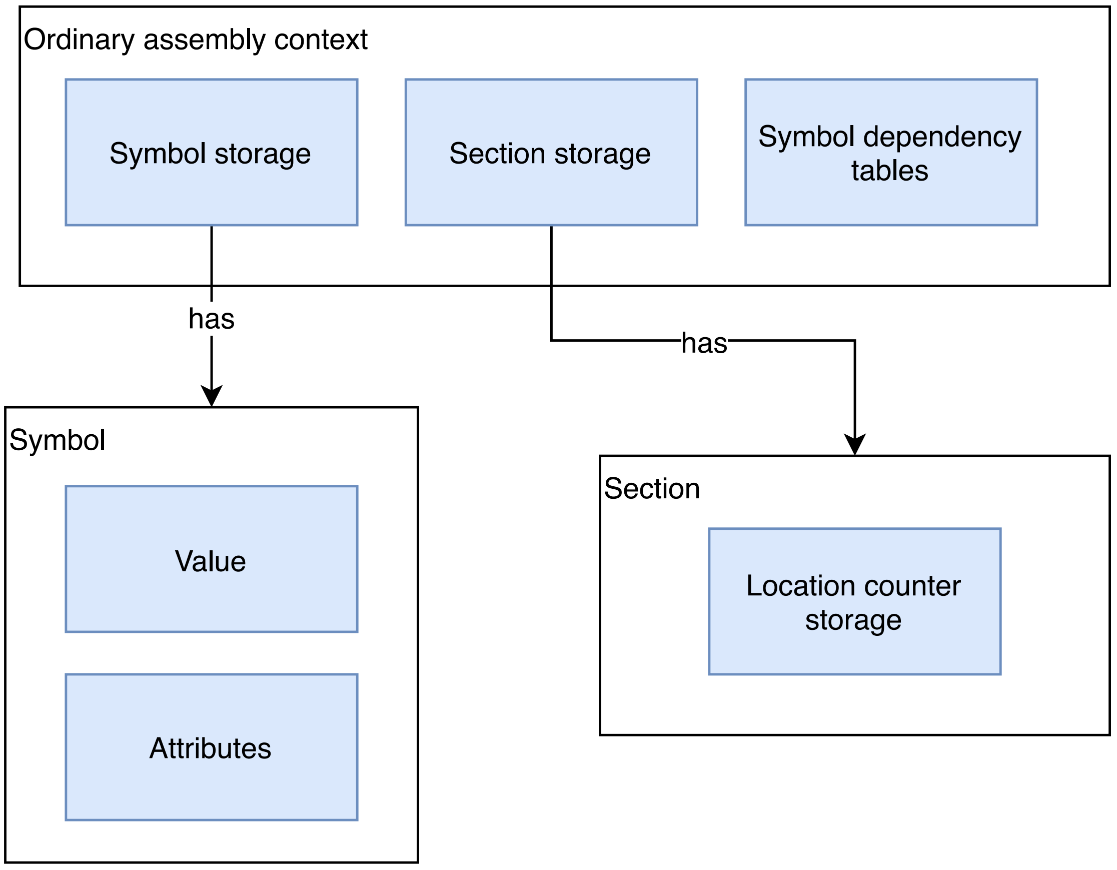

Figure 6.8: The composition of ordinary assembly context

describing structures get complicated. Therefore, HLASM context contains object storing just this part of the processing.

Ordinary assembly context consists of three main components (see fig. 6.8):

1. *Symbol storage*. Stores ordinary symbols (see section 2.2.1.1).
2. *Section storage*. Has notion of all generated sections, each section containing its location counters.
3. *Symbol dependency tables*. Contains yet unresolved dependencies between symbols prior to the currently processed instruction.

6.7.8.1 Symbol

`parser_library/src/context/-ordinary_assembly/symbol.h`
`parser_library/src/context/-ordinary_assembly/symbol_attributes.h`

This class represents HLASM ordinary symbol (see section 2.2.1.1). Besides its identifier and location, symbol contains *value* and *attributes* components.

Value can be assigned *absolute* or *relocatable* values. With addition to that, it can also be assigned an empty value stating that symbol is not yet defined.

Attributes structure holds symbol attributes like type, length, scale and integer.

6.7.8.2 Section

`parser_library/src/context/-ordinary_assembly/section.h`

Section is a structure representing HLASM section (created by CSECT, DSECT, ...). It contains enumeration *section kind* describing type of the section prior to the used instruction. The structure also holds *location counter storage* with defined location counters.

6.7.8.3 Location counter

`parser_library/src/context/-ordinary_assembly/location_counter.h`
`parser_library/src/context/-ordinary_assembly/location_counter_data.h`
`parser_library/src/context/-ordinary_assembly/address.h`

This structure contains data and operations for one location counter. The data is stored in helper sub-structure *location counter data*.

Location counter data is a structure defining current value of the location counter. It consists of:

- *Storage* stating total number of bytes occupied by the location counter.
- Vector of *spaces*, blocks of bytes with yet not known length.
- Vector of *storage* between each space.
- Currently valid *alignment* (used when data contain spaces).

The location counter value is transformable into a relocatable value. It is represented by structure *address*.

Address consists of:

- Array of *bases*. A base is a beginning of a corresponding section. They serve as points of reference for the address.
- Array of *spaces* that are present in the address.
- *Offset* from the bases.

The common composition of an address is one base section (as the start of the address) and value of storage (as the offset from it).

The need for the whole array of bases to be present is because addresses from different sections can be arbitrarily added or subtracted. This information is needed as the correct sequence of arithmetic operations can reduce number of bases (even spaces) to zero and create absolute value. This value can be later used in places where a relocatable value would be forbidden.

Space is block of bytes with yet not known length. It is created in the active location counter when execution of counter's operation can not be performed due to non previously defined ordinary symbols. See the different kind of spaces and the reason of creation in the table 6.7.

Space Kind	Creation
Ordinary	when instruction outputs data of unknown length
LOCTR begin	when defining more than one location counter in a section
Alignment	when current alignment is unknown due to previous spaces
LOCTR set	when moving counter's value to the address with spaces
LOCTR max	when moving counter's value to the next available location
LOCTR unknown	when moving counter's value to the yet unknown address

Table 6.7: Different kinds of spaces and reasons of creation.

When a space length becomes known, all addresses containing the spaces need to be updated (remove the space and append offset). Therefore, space structure contains an array of address listeners. Hence, when an address is assigned a relocatable value that contains the space, the address is added to its array. This serves as an easy point of space resolving.

ORG instruction can arbitrarily move location counter's value forward and backward. With addition to that, ORG can also order location counter to set it's value to the next available value (the lowest untouched address, see section 2.2.2.2). Combining this with the possible spaces creation, location counter holds an array of the location counter data to properly set the next available value.

6.7.8.4 Symbol dependency tables

`parser_library/src/context/-ordinary_assembly/symbol_dependency_tables.h`
`parser_library/src/context/-ordinary_assembly/dependant.h`
`parser_library/src/context/-ordinary_assembly/dependable.h`

HLASM forbids cyclic symbol definition. This component maintains dependencies between symbols and detects possible cycles. Let us describe the main components of dependency resolving.

Dependant is a structure used in the symbol dependency tables. It encapsulates objects that can be dependent on another. Dependant object can be a *symbol*, *symbol attribute* and *space*.

Dependable interface is implemented by a class if its instance can contain dependencies. The interface has a method to retrieve a structure holding the respective *dependants*.

Resolvable interface adds up to the dependable interface. It is implemented by objects that serve as values assignable to *depednants*. It provides methods to return *symbol value* with help of the dependency solver.

Dependency solver is an interface that can return value of the symbol providing its identifier. It is implemented by Ordinary assembly context.

Having described building blocks, we can move to the symbol dependency tables composition.

Dependency map is the primary storage of dependencies. It has *dependants* as keys and *resolvables* as values. The semantics for pair *(D,R)* is that D is dependent on the dependencies from R. Each time new dependency is added, this map is searched for cycle.

Dependency sources map serves as a source objects storage of a resolvable in the dependency map. Hence for the pair *(D,R)* from dependency map, source object of *R* is in the dependency source map under the key *D*.

The source objects are statements. To be more specific, as one statement can be a source for more distinct resolvables, this source map only stores pointers to the *postponed statements storage*.

Postponed statements storage holds statements that are sources of resolvables in dependency map. The reason they are stored is that they can not be checked yet as they contain dependencies. Therefore, they are postponed in the storage until all of the dependencies are resolved. Then they are passed to the respective checker.

7. Macro Tracer

The macro tracer allows the user to track how the HLASM source code is assembled in experience similar to common debugging tools. The user is able to see step by step how CA instructions are interpreted and how macros are expanded.

This is achieved by implementing the Debug Adapter Protocol. The protocol itself is implemented in the language server component, which uses the macro tracer component.

7.1 DAP functionality mapping

The DAP was originally designed to communicate between an IDE or an editor and a debugger or a debug adapter. For example, when debugging a C++ application in Visual Studio Code, the editor communicates through DAP with a debugger that is attached to a compiled C++ application. Contrary to this, the macro tracer does not run with compiled binary, it only uses the analyzer to simulate the compilation process of high level assembler.

However, even though we are not implementing a real debugger, it makes very good sense to use a debugging interface for tracing the simulation. Parts of the debugging interface that we use in a macro tracer are as follows.

Instruction pointer The instruction pointer is commonly shown in debuggers by highlighting a line of code that is going to be executed next. This is applicable to HLASM without change, since all the instructions are processed one by one in a well-defined order.

Breakpoints The user can set a breakpoint when he is interested in tracing only particular section of the code. The compilation simulation will stop when it reaches a line with a breakpoint.

Continue The user can restart a paused simulation by using the continue function just as in any debugger.

Step in and step over In debuggers, it is possible to use step in / step over functions to debug an implementation of subroutine or to skip it and continue after the application returns from the subroutine. In HLASM, this can be applied to macros and COPY instructions: if the user is interested in what happens inside a macro or a COPY file, he can use step in. Step over skips to the next instruction in the same file.

Variables The same way common debuggers show values of runtime variables, the macro tracer uses the same functionality to show values of set symbols, macro

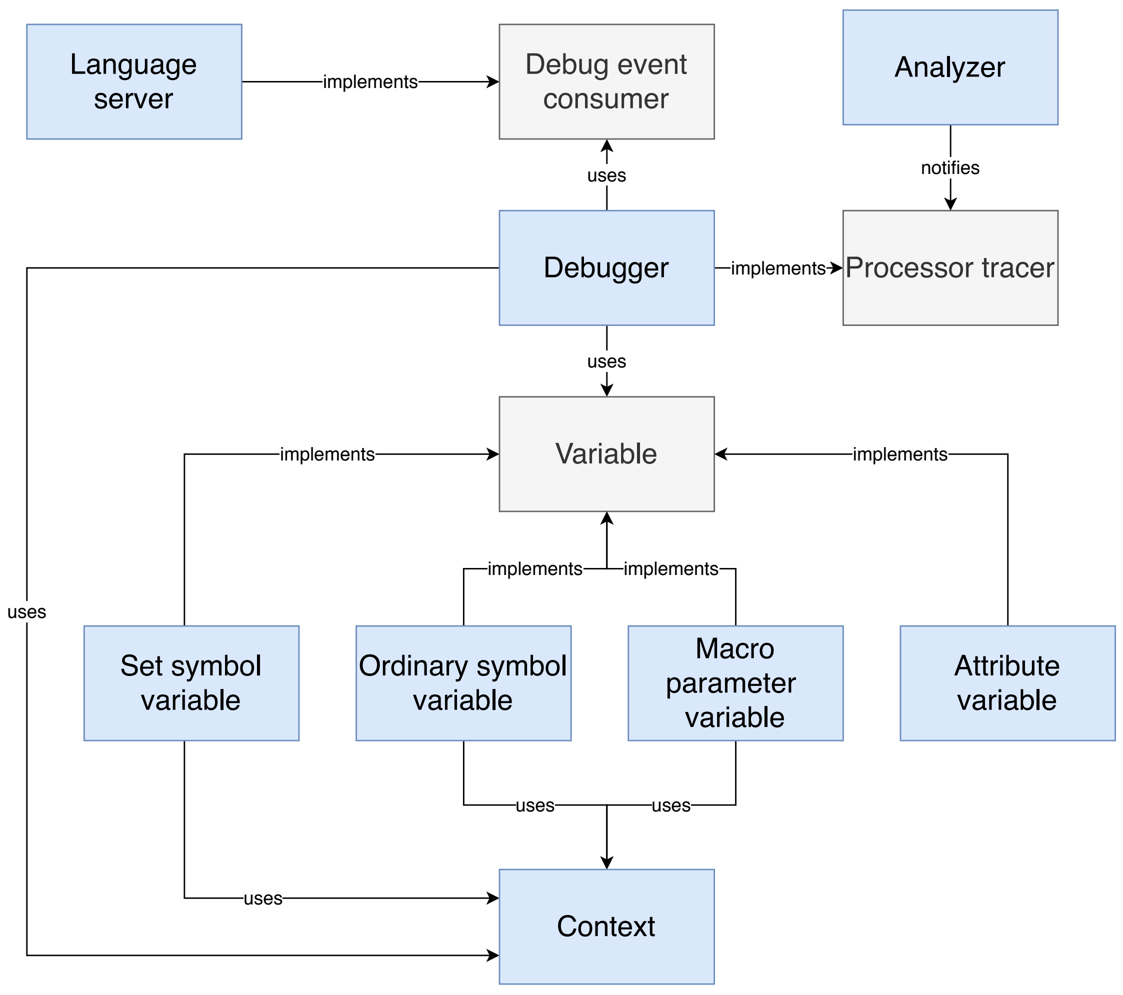

Figure 7.1: Architecture of the macro tracer

parameter values and ordinary symbols. It is also possible to visualize attributes of symbols.

Call stack The call stack makes sense with the macro tracer too. It can show the stack of currently processed macros and COPY files. Moreover, macros have local set symbols and parameters, so each stack frame may show a different set of valid variables.

All described functionality (and more) is supported by the DAP.

7.2 Macro tracer architecture

parser_library/src/debugging/variable.h

The macro tracer architecture is shown in fig. 7.1.

`Debugger` is a class that encapsulates all macro tracer functionality. It starts the standard analysis provided by the `analyzer` component in a special thread. The `debugger` implements the processor_tracer interface, which allows it to receive a notification every time a statement is about to be processed.

It is also the debugger's responsibility to extract data from the context used by the analyzer and to transform them into a form compatible with the DAP.

Debugger uses an interface variable which represents the variable as it is shown to the user — most importantly, it is a name-value pair. The variable interface has four implementations:

- set_symbol_variable parser_library/src/debugging/set_symbol_variable.h
- ordinary_symbol_variable parser_library/src/debugging/ordinary_symbol_variable.h
- macro_parameter_variable parser_library/src/debugging/macro_param_variable.h
- attribute_variable parser_library/src/debugging/attribute_variable.h

First three represent a HLASM symbol of respectable type. They adapt the context representation of the symbols to DAP variables.

The attribute_variable represents attributes of all types of symbols. It does not access context, and it is only used by the rest of variables to show their attributes.

7.3 Debugger

parser_library/src/debugging/debugger.h
parser_library/src/debugging/debug_types.h
parser_library/src/processing/processing_tracer.h

The debugger component is the core of the macro tracer implementation. When the user starts debugging, the method launch is called from the language server component. The debugger creates analyzer and starts the analysis in a separate thread. The debugger implements processor_tracer interface, which only has one method — statement. The analyzer calls the statement method every time a next statement is about to be processed.

This implementation makes it possible for the debugger to stop the analysis using a conditional variable. When it sees fit (e.g. when a breakpoint was hit), the debugger can put the thread to sleep and wait for further user interaction. At the same time, it notifies the language server through debug_event_consumer interface that the analysis has stopped.

There are three important structures in the DAP:

- **Stack frame** Stack frame represents one item in the call stack. Each frame has a name that is shown to the user and points to a line in the source code. In the macro tracer, each frame points either to the next instruction, to a macro call or to a COPY instruction.
- **Scope** Each stack frame may have scopes. A scope is simply a group of variables used to make them organized for the user. The macro tracer uses three scopes: local variables, global variables and ordinary symbols.
- **Variable** Each scope has arbitrary number of variables. Each variable has a name and a value. They may be further structured and may have additional child variables. Therefore, the DAP can be used to present arbitrary tree of variables to the user. Listing 12 shows an example regarding nested macro parameters.

Listing 12 An example of how the macro tracer leverages DAP nested variables. First line shows a macro call with a parameter. HLASM treats such parameters as nested arrays. Second part shows how such a parameter is shown in VS Code using nested variables.

```
MAC (foo,((bar,ex),am),ple,(lorem,ipsum))

1: foo
2: ((bar,ex),am)
  1: (bar,ex)
    1: bar
    2: ex
  2: am
3: ple
4: (lorem,ipsum)
  1: lorem
  2: ipsum
```

While the thread is stopped, the editor sends requests to display information about the current context. It is the `debugger's` responsibility to extract a list of stack frames from the context, return a list of scopes for each stack frame and a list of variables for each scope. It does not have to deal with the complexity of different types of set symbols and macro parameters, which is done by the implementations of the `variable` interface.

8. VS Code extension

The frontend of the project is implemented as an extension to a modern IDE instead of creating a completely new GUI. This approach has the advantage of providing a familiar environment and workflow that the developers are used to.

There are several IDEs that currently (natively) support LSP, such as *Eclipse Che* and *Eclipse IDE*, *vim8*, *Visual Studio* and *Visual Studio Code* and many more. Others, for example *IntelliJ*, have plugins which add the support for LSP.

Our IDE choice is *Visual Studio Code*, due to its popularity and lightweight design. Conveniently, *Theia*, a web-based IDE, supports VSCode extensions, therefore our plugin works with *Theia* as well.

8.1 Standard LSP Extension

`clients/vscode-hlasmplugin/src/extension.ts`

The core of the extension is an activation event which starts the plugin for VSCode.

Upon activation, *Language Client* and *Language server* 4 are started as child processes of VSCode and a pipe is open for their communication. The LSP communication and its features are handled by the *vscodelc* package.

To be independent of pipes, we have added an option to use TCP, which assigns a random free port for TCP communication.

8.2 DAP Extension

Macro Tracer 7 is implemented using DAP, which is also supported out-of-the-box by VSCode. Similarly to LSP TCP support, we dynamically assign a random free port for DAP communication during the activation.

8.3 Additional implemented features

To simplify the work with HLASM in modern editors, several features are added to the extension . These additions are specific for Visual Studio Code (and Theia) and are not a part of the LSP specification.

8.3.1 Language Detection

The usual workflow with the extension begins with downloading HLASM source codes from mainframe. Typically, these files will not have any file extension and even if they do, they might differ across various products.

To cope with this problem, there are several mechanisms that help the user to recognize the file as HLASM automatically.

Macro Detection Each file starting with line *MACRO* (arbitrary number of whitespace before and after) is recognized as HLASM.

Configuration Files Detection Every file either defined as a program or as a part of a processor group is recognized as HLASM.

Wildcards Configuration file *pgm_conf.json* contains a field *alwaysRecognize*, which consists of user-defined wildcards. Every file that satisfies at least one of these wildcards is recognized as HLASM.

Automatic Language Detection Whenever a user opens a file, its contents are scanned line by line. If the file has a sufficient ratio of HLASM lines to all lines, it is considered to be HLASM.

The HLASM line recognition is mostly based on a pre-defined set of most used instructions. If a line correctly uses one of these instructions, it is counted as a HLASM line. Continued line of a HLASM line is also a HLASM line. Moreover, a HLASM line must not exceed 80 characters.

Comment lines or empty lines are skipped and not counted.

We tested the detection on 11.000 HLASM files and 9.000 non HLASM files. The best results were observed using 4/10 ratio, with 88% true positive recognition and 95% true negative recognition.

Because of the indeterminate outcomes, this method is meant to be used as a fallback in case all previous methods do not suffice.

All detection layers are visualized in fig. 8.1.

8.3.2 Continuation Handling

Due to historical reasons, HLASM has a 80 character-per-line limitation. Modern languages do not enforce such restriction and therefore IDEs such as VSCode allow the user to extend their lines freely. This causes 2 major inconveniences.

First of all, the user must add the continuation character on a very specific column manually. Secondly, each time the user types in between continuation character and the instruction/parameters, the continuation character is pushed from its requisite position and needs to be moved back, again manually.

To improve this behavior, the extension offers an option to activate *Continuation Handling*.

The first problem is solved by adding two editor commands *insertContinuation* and *deleteContinuation*, which, when invoked, insert/delete the continuation character on its correct position.

To improve the second problem, the option overrides standard VSCode commands, commonly used when working in editor such as *type*, *deleteLeft*, *deleteRight*, *cut* and *paste*. They offset the continuation character by removing/adding whitespaces in front of it.

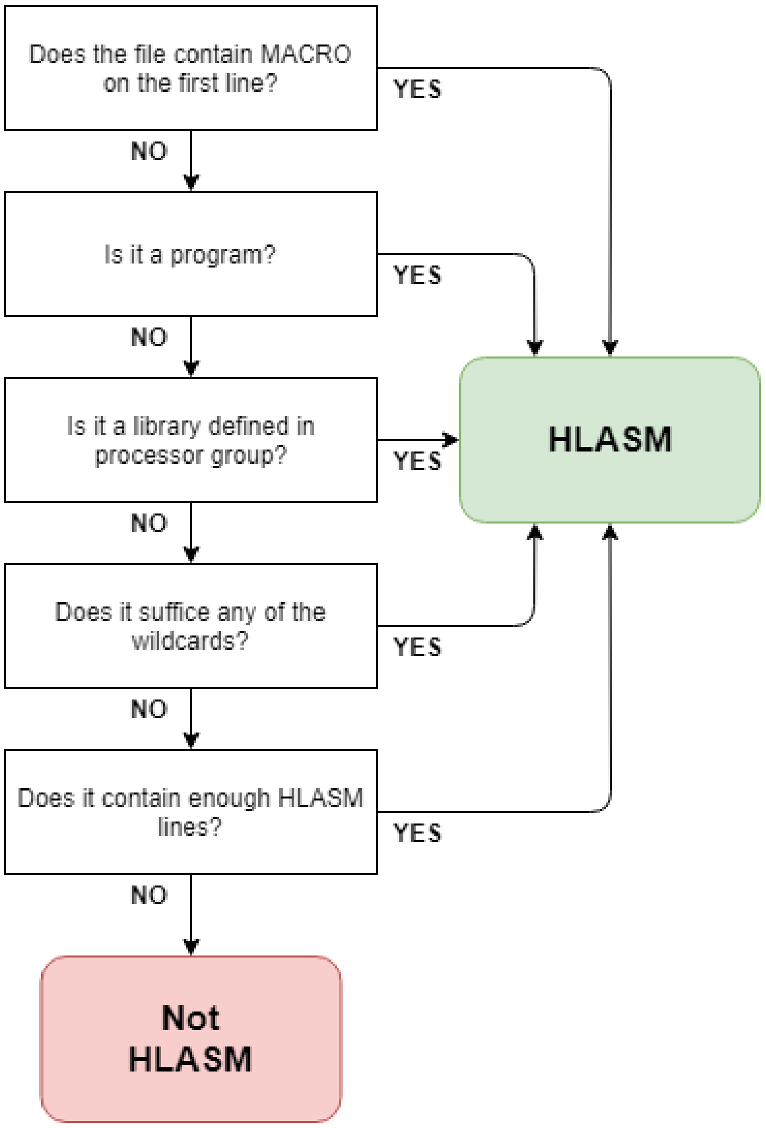
Does the file contain MACRO on the first line?
YES
NO
Is it a program?
YES
NO
Is it a library defined in processor group?
YES
NO
Does it suffice any of the wildcards?
YES
NO
Does it contain enough HLASM lines?
YES
NO
HLASM
Not HLASM

Figure 8.1: Language Detection layers.

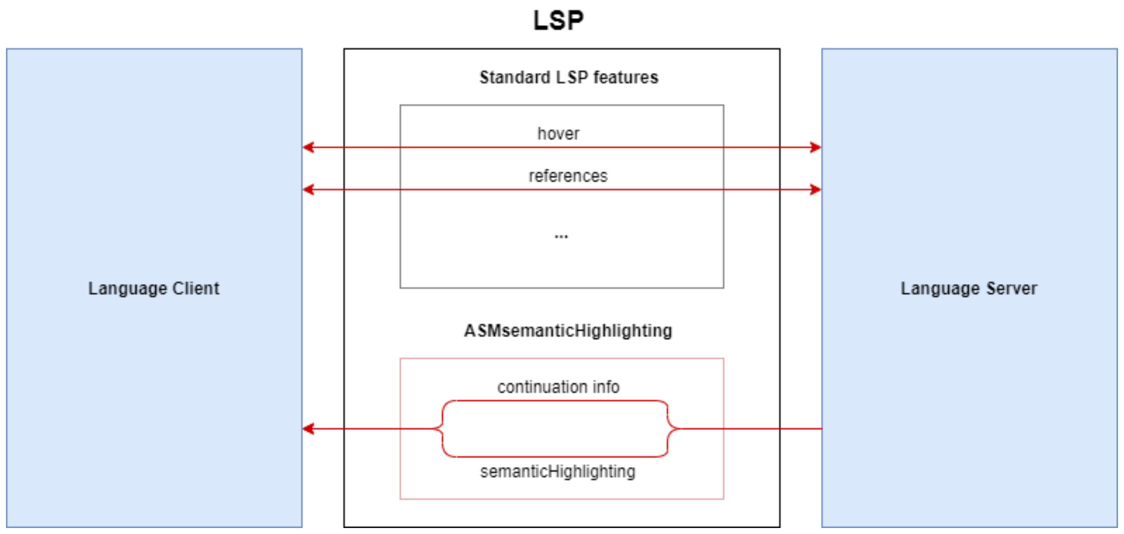

Figure 8.2: The addition of semantic highlighting to the LSP communication.

8.3.3 Configuration Prompt

If a workspace contains a HLASM file, but does not have the configuration files set, the user is prompted to create them. The warning message also offers an option to create templates for them.

8.3.4 HLASM Semantic Highlighting

```
clients/vscode-hlasmplugin/src/-
 semanticHighlighting.ts
clients/vscode-hlasmplugin/src/-
 ASMsemanticHighlighting.ts
clients/vscode-hlasmplugin/src/-
 protocol.semanticHighlighting.ts
```

In case of HLASM, a semantic (server-side) highlighting is desired. The multi-layered nature of the language causes that in quite common scenarios, specific parts of the code can be properly highlighted if and only if some previous part was completely processed (parameters for instructions, skipped code thanks to code generation, defined macros, continuations, etc...).

Based on the open pull request to the VSCode Language Server [1], we added *semanticHighlighting* as an extra feature of LSP. This feature works in a very similar manner, implementing the LSP interfaces that VSCode provides. It works as a notification from the server to the client, containing ranges inside the document and their respective tokens (e.g. instruction, label, parameter, comment,..).

On top of that, we extended *semanticHighlighting* to *ASMsemanticHighlighting*, which adds the ability to notify the client about a new code layout, specifically begin, continuation and continue columns. These fields can be set in the HLASM code (via ICTL instruction) and are required for the *Continuation Handling* feature to work properly. Our client-server communication is shown in fig. 8.2.

[1]https://github.com/microsoft/vscode-languageserver-node/pull/367/files

Part III

Dependencies and build guide

9. Third party libraries

The project uses several third party libraries. Some of them are needed by language server to parse LSP messages and communicate through DAP. Also, we use a third party library to recognize syntax of HLASM.

ASIO C++ library[1] Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach. We use it to handle TCP communication in a cross-platform way. Asio implements std::iostream wrappers around the TCP stream, which allows us to abstract from the actual source of the communication.

JSON for Modern C++[2] We use JSON for Modern C++ library to parse and serialize JSON. It is used in both LSP and DAP. It allows us to seamlessly traverse input JSON and extract the interesting values, as well as easily respond with valid JSON messages.

cpp-netlib URI[3] Cpp-netlib URI library is used for parsing URI specified by the RFC3986[4], which is used by the LSP and DAP protocols to transfer paths to files. It is the responsibility of the language server to parse the URIs and convert them to file paths, so it is easier to work with them in the parser library.

9.1 ANTLR 4

We have based part of our analyzer on ANTLR 4 parser generator. ANLTR 4 implements Adaptive $LL(*)$ [Parr et al., 2014] parsing strategy.

9.1.1 Adaptive $LL(*)$ parsing strategy

Adaptive $LL(*)$ (or short $ALL(*)$) parsing strategy is a combination of simple, efficient and predictable top-down $LL(k)$ parsing strategy with power of GLR which can handle non-deterministic and ambiguous grammars. Authors move the grammar analysis to parse-time. This lets $ALL(*)$ handle any non-left-recursive context-free grammar rules and for efficiency it caches analysis results in lookahead DFA.

Theoretical time complexity can be viewed as a possible downside of $ALL(*)$. Parsing of n symbols takes $O(n^4)$ in theory. In practice, however, $ALL(*)$ seems to outperform other parsers by order of magnitude.

[1] https://think-async.com/Asio/

[2] https://github.com/nlohmann/json

[3] https://github.com/cpp-netlib/uri

[4] https://tools.ietf.org/html/rfc3986

Despite the theoretical $O(n^4)$ time complexity, it appears that the $ALL(*)$ behaves linear on most of the code, with no unpredictable performance or large footprint in practice. In order to support this, authors investigate the parse time vs file size for languages `C`, `Verilog`, `Erlang` and `Lua` files. They found very strong evidence of linearity on all tested languages (see the original paper for details).

9.1.2 ANTLR 4 pipeline

ANTLR 4, similar to any other conventional parser generator, processes the inputted code as follows: (1) breaks down the source string into tokens using *lexer* (2) *parser* build parse trees.

This pipeline in ANTLR 4 is broken into following classes:

`CharStream` represents input code.

`Lexer` breaks the inputted code into tokens.

`Token` token representation that includes important information like token type, position in code or the actual text.

`Parser` builds parse trees.

`TokenStream` connects the lexer and parser.

fig. 9.1 sketches the described pipeline.

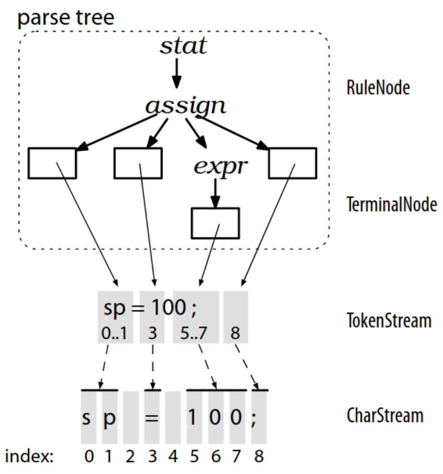

Figure 9.1: ANTLR 4 pipeline overview. Taken from [Parr, 2013].

9.1.3 ANTLR Parser

The input to ANTLR is a grammar written in antlr-specific language that specifies the syntax of HLASM language. The framework takes grammar 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 and produce helper structures (statements).

9.1.4 Parse-Tree walking

We employ the ANTLR 4 Parse-Tree Visitors during the expression evaluation. ANTLR 4 offers two mechanisms for tree-walking: the parse-tree listeners and parse-tree visitors. As the tree-walker encounters a rule, it triggers a *start* function. Similarly, when the walker visits all children of the node, it calls the *finish* function. The second mechanism is the parse-tree visitor. The visitor lets the programmer control the walk by explicitly calling methods to visit children. We picked the latter approach because we need to have ampler control over the evaluation (such as operator priority).

9.1.5 Visitor

We employ ANTLR *visitor* feature during evaluation of CA expressions.

The ANTLR 4 first generates `hlasmparserVisitor` and `hlasmparserBaseVisitor`. The former is a abstract class, the latter is a simple implementation of the former. Both classes define "visit" functions for every grammar rule. A visit function has exactly one argument — the context of the rule. The simple implementation executes `visitChildren()`. Our parse-tree visitor — the `expression_evaluator` — overrides `hlasmparserBaseVisitor`. In order to evaluate a sub-rule, we call `visit(ctx->sub_rule())`, where `ctx->sub_rule()` returns the context of the sub-rule. The `visit()` function matches appropriate function of the visitor based on the context type (for example, `visit(ctx->sub_rule())` would call `visiSub_rule(..)`).

10. Build instructions

In this chapter, we describe how to build the project on different platforms. We only describe methods that we use and are guaranteed to work, but other platforms and versions may work as well.

The result of a build is the Visual Studio Code extension packed into a VSIX file, which can be found in the `bin/` subdirectory of the build folder.

10.1 Prerequisites

In order to build the project on any platform, following software needs to be installed:

- CMake 3.10 or higher
- C++ compiler with support for C++17
- Java Development Kit (JDK) 8 or higher (the ANTLR project written in Java is built from sources)
- Maven (the build system of ANTLR)
- Git (needed to download sources of the third party software)
- npm (for compiling the typescript parts of the VS Code extension)

10.2 Windows

On windows, we use Visual Studio Community 2019. We also have VS configurations for building and testing the project in WSL.

It is also possible to build the project from command line:

```
mkdir build && cd build
cmake ../
cmake --build .
```

10.3 Linux

In addition to the prerequisites listed in section 10.1, linux build has two more prerequisites:

- pkg-config
- UUID library

We build the project for Ubuntu 18.04 and for the Alpine linux.

10.3.1 Ubuntu

On Ubuntu 18.04 the following commands install all prerequisites and then build the project into `build` folder:

```
apt update && sudo apt install cmake g++-8 uuid-dev npm default-jdk
                       pkg-config maven
mkdir build && cd build
cmake -DCMAKE_C_COMPILER=gcc-8 -DCMAKE_CXX_COMPILER=g++-8 ../
cmake --build .
```

10.3.2 Alpine linux

The build works on Alpine linux version 3.10. The following commands install all prerequisites and then build the project into `build` folder:

```
apk update && apk add linux-headers git g++ cmake util-linux-dev npm ninja
                      pkgconfig openjdk8 maven
mkdir build && cd build
cmake ../
cmake --build .
```

10.4 Mac OS

We have only built the project on MacOS 10.14. In order to successfully build, we require LLVM 8 (it can be installed by using homebrew).

The project can be built with a snippet like this:

```
mkdir build && cd build
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
      -DLLVM_PATH=<path-to-llvm-installation> ../
cmake --build .
```

For instance, a possible path to LLVM is `/usr/local/opt/llvm8`

10.5 Running tests

Once the project is built, there are two test executables in the `bin/` subdirectory: `library_test` and `server_test`. Just run both of them to verify the build.

Bibliography

[Parr, 2013] Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[Parr et al., 2014] Parr, T., Harwell, S., and Fisher, K. (2014). Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598.

Source file documentation index

Appendices

A. Architecture of the project

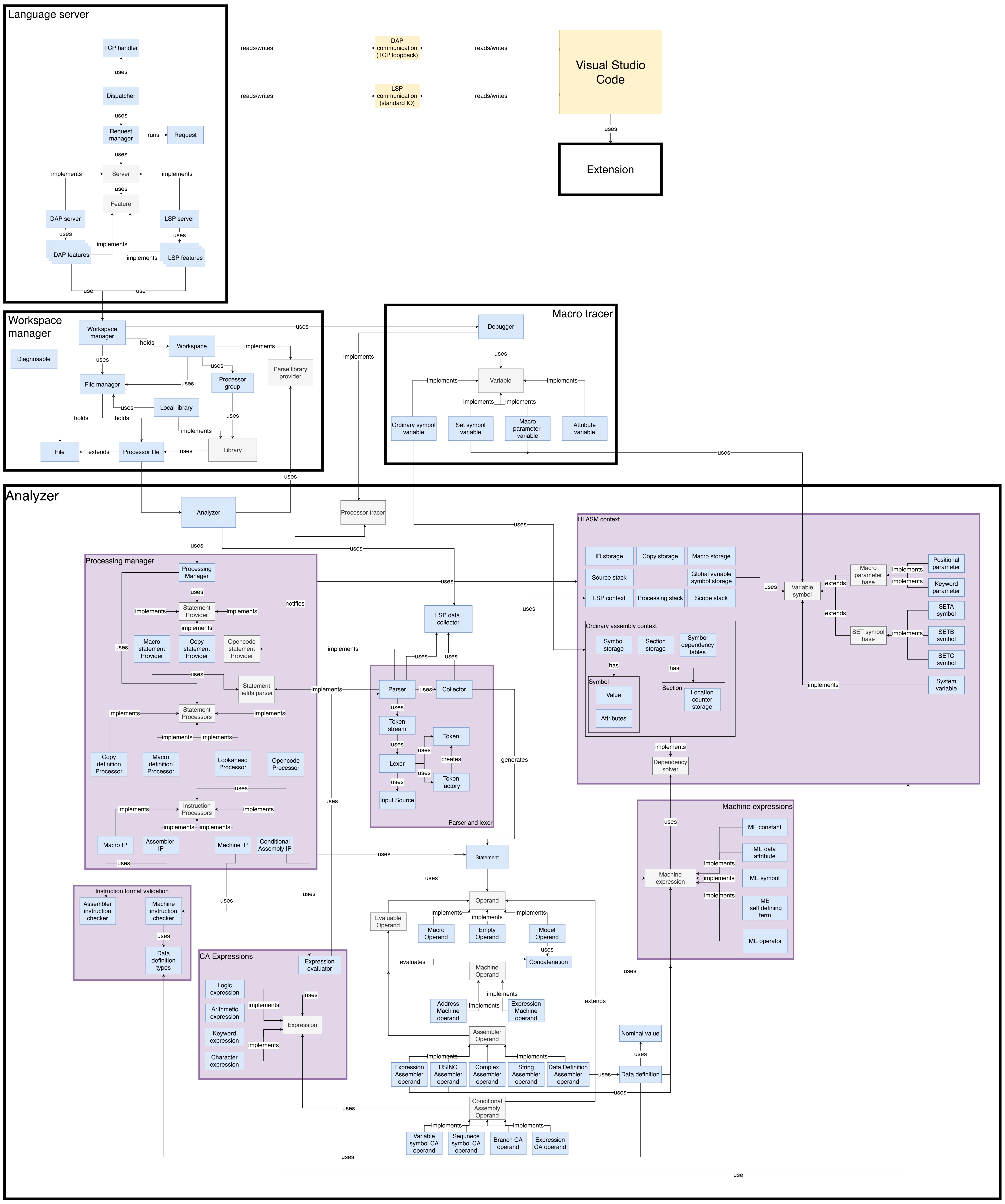

Figure A.1: Architecture of the whole project

B. Parser grammar

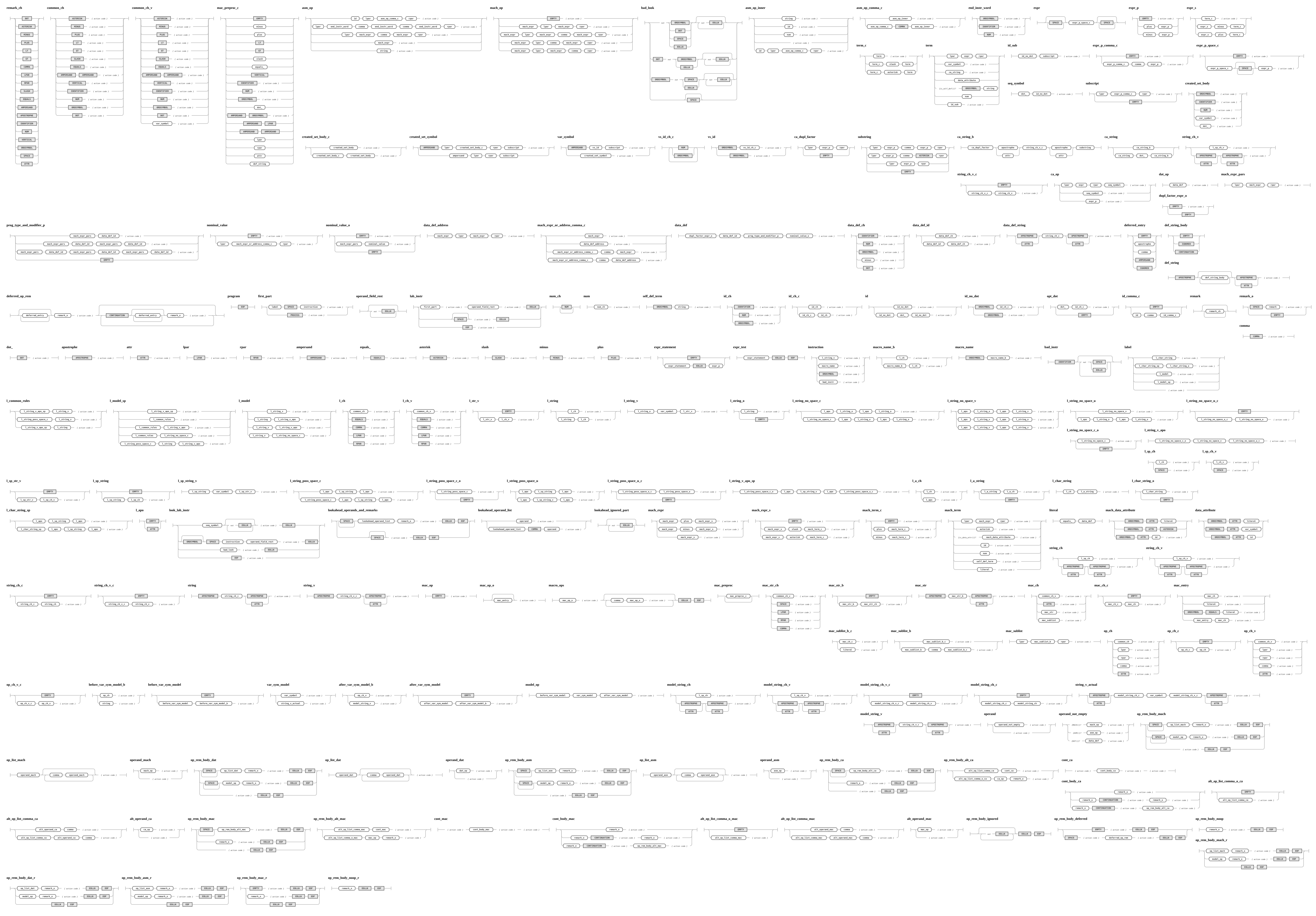

Figure B.1: All 193 grammar rules