High Level Assembler Plugin
Project specification

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

Supervisor: Miroslav Kratochvíl

Contents

1. Background and goals

1.1 Related Work

misto ‘related work’ je tady vhodny mit spis ‘related HLASM users’

2. HLASM overview

In general, high-level assemblers provide for their assembly languages features that are commonly found in high-level programming languages. Hence, in addition to ordinary machine instructions they also contain control statements similar to *if, while, for* as well as custom callable macros.

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

2.1 Syntax

Because of historical reasons HLASM syntax is fairly complicated. Its line length is limited to 80 characters as it was in times when punch cards were used.

Besides this HLASM uses syntax common to regular assemblers.

2.1.1 Statement

HLASM program is sequence of *statements*. Statement consists of four fields. Those are:

- **Name field** — Serves as a place for named constants that are to be used in code. The field is optional but when present it must start in the begin column of a line.
- **Operation field** — Instruction that is executed. The only field that is mandatory. Must not begin in the first column as it would be interpreted as a name field.
- **Operands field** — Field for instruction operands separated by comma located immediately after operation field. According to instruction used it can be any sequence of characters, apostrophe separated string or blank.
- **Remark field** — Serves as inline commentary. Optionally located after operands field or operation field when operands are blank.

This is an example of basic statement using all field.

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

2.1.2 Continuation

One line in HLASM source code can contain only up to 80 characters. However, sometimes statement is too long to be written in one line. Therefore, special handling is introduced called **continuation**.

Prosim nepouzivejte bold uprostred odstavce nebo v textu, na emphasis a definice je *emph*. Pokud je neco potreba zvyraznit, je to potreba udelat systematictejc, idealne obrazkem.

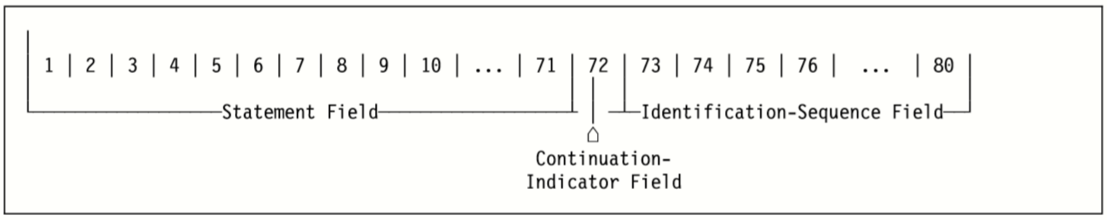

Figure 2.1: Description of line columns (source HLASM Language Reference https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3sc264940/-$file/asmr1023.pdf).

Firstly, let us elaborate more on the topic of line column. There are four special columns:

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

They all serve different purpose. *Begin column* states start of the statement or where name field should be written. Anything after *end column* does not count as the content of a statement, rather it is used as a place for the line sequence number (see 2.1).

Continuation column is used for indication that statement continues on the next line (to correctly indicate we write there arbitrary character other than space). Then the remainder of the statement must start on *continue column* to finally create a well formed statement.

Here is an example of an instruction where its last operand exceeded 72. column of the line.

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

However, there are some instructions that allow so called *extended format* of operands allowing continuation even when the contents of a line have not reached the continuation column.

reference the figure, do not use [h].

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

2.2 Assembling

Having briefly described syntax, this section prepares reader to better understand assembly process hidden behind HLASM.

We can divide assembling into two interlinked steps, **conditional assembly** and **ordinary assembly**.

2.2.1 Conditional assembly

This part of assembly process can be compared to C++ text prepocessor. In HLASM it is more complicated process so it has obtained the term *code generation*. It consists of **variable symbols**, **conditional assembly (CA) instructions** and **macros**.

2.2.1.1 Variable symbols

These symbols serve as points of substitution or information holders.

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

Variable symbols have also notion of types. Symbol can be integer, boolean or string. CA instructions gather this information for different sorts of conditional branching.

2.2.1.2 CA instructions

The major difference to other instructions is that they are not assembled into object code, they rather select which instructions will be processed by assembler next.

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

Other subset is capable of conditional and unconditional branching. HLASM provides big variety of built in binary or unary operations on variable symbols which can create complex conditional expressions. This is important in HLASM as you can alter flow of instructions that will be assembled into executable program.

2.2.1.3 Macros

Macro is structure consisting of name, input parameters and sequence of statements called body. When they are called in HLASM program, each statement in the body is performed. Nested or recursive call of macros is allowed. Macro body can even contain such sequence of instructions that it can generate another macro definition ready for later use. With help of variable symbols, HLASM macros have power to create custom task specific macros.

2.2.2 Ordinary assembly

Ordinary assembly is a term for assembly other that conditional.

Assembly of *machine instructions* belong here. They and their operands are translated to sequence of bytes and written to executable program. HLASM differs from basic assemblers as it allows expressions as operands of those instructions. This expressions can contain constanst as well as are capable of address arithmetics.

Assembly of *assembler instructions* also belong here. However, they are neither assembled nor completely ignored. They alter behavior of assembler.

2.2.2.1 Assembler instructions

The behavior of assembler is altered by this instructions in different ways. Let us enumerate some of them.

- **ICTL** — Changes previously described line columns (i.e. *begin column* at column 2 etc.).
- **DC** — Reserves space in object code for data described in operands field and assembles them in place (i.e. assembles float, double, character array, address etc.).
- **EQU** — Defines named constant with integer value or relative address value. This constants can be accessed by *conditional assembly*, hence alter it in custom manner.
- **COPY** — Copies whole file found in *copy member library*[1] and pastes it in place of the instruction.
- **CSECT** — Creates an executable control section. Serves as the beginning of a machine instruction sequence and start of relative addressing.

Here is example of simple HLASM program with the description of its statements.

```
        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          D'-3.729'
[18]    .END        ANOP
[19]                END
```

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

In line `06` there is use of *copy instruction* where it includes contents of `REGS` file.

Line `08` establishes start of executable section called `TEST`.

In line `09` integer value is assigned to variable symbol `VAR`. The value is the length attribute of non previously defined constant `DOUBLE`. The assembler looks for definition of the constant to properly evaluate conditional assembly expression. In the next line there is CA branching instruction `AIF`. If value of `VAR` equals 4, next lines are skipped and assembling continues on line `18` where branching symbol `END` is located.

[1] Path to library is passed to assembler before the start of assembly.

Lines `12-13` shows example of machine instructions which are directly assembled into object code. Lines `11,14` are examples of macro call.

In line `15` to constant `LEN` is equated difference of two addresses. This value is next used to generate character data.

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

`ANOP` is empty assembler action and line `19` ends the program assembling.

As the reader may see, HLASM is heavily extended assembler with complex assembling phases. However, the result of that is programming language with large expressive power.

3. Requirements

-co ten nas produkt ma byt vseobecne zhrnutie

...je to extension ... doda support pre ... -cela tato sekcia uz je popisana niekde na CA wiki, mozno dobry zaklad

Tohle by mozna nebylo spatny rovnou pojmenovat nejak jako 'Features', 'API' nebo mozna 'Interfaces'.

3.1 Language features

-zoznam veci jazyka co podporujeme

3.2 LSP features

-working plugin for vs code

- Go to definition for all symbols, macro definitions and copy members.
- Find all references
- Completion for instructions, defined symbols and macros
- Highlighting
- Hover

-non functional requirement - api kniznice??

4. Architecture

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

In LSP the two parties who communicate are called *client* and *language server*. A simple example is displayed in Figure fig. 4.1 The client runs as a part of an editor. Language server may be standalone application, that is connected to the client by a pipe or TCP. All language-specific user actions are transformed into standard LSP messages and sent to the language server, which analyses the source code and sends back response, which is then interpreted and presented to the user in editor-specific way. This architecture makes possible to have only 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 LSP client.

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

This chapter presents decomposition of the project into smaller components and describes their relations. The main two components are language server — an executable application that uses parser library. The overall architecture is pictured in Figure fig. 4.2.

4.1 Language server

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

- To read LSP messages from standard input or TCP and write responses.
- To parse JSON RPC to C++ structures so they can be further used.
- To serialize C++ structures into JSON, so it can be sent back to client.
- Implement asynchronous request handling: e.g. when user makes several consecutive changes to a source code, it is not needed to parse every change, only the final version.

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

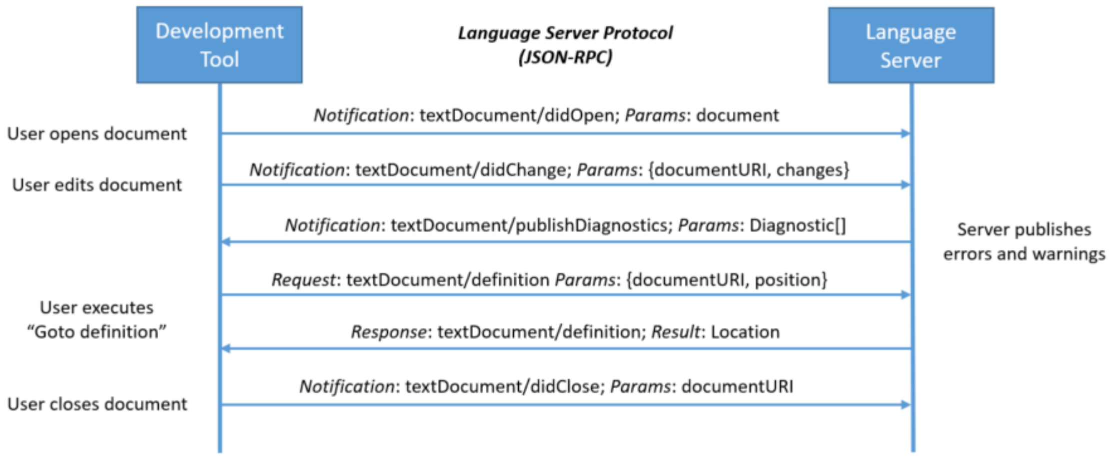
Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; *Params*: document
User edits document
Notification: textDocument/didChange; *Params*: {documentURI, changes}
Notification: textDocument/publishDiagnostics; *Params*: Diagnostic[]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

Figure 4.1: LSP session example

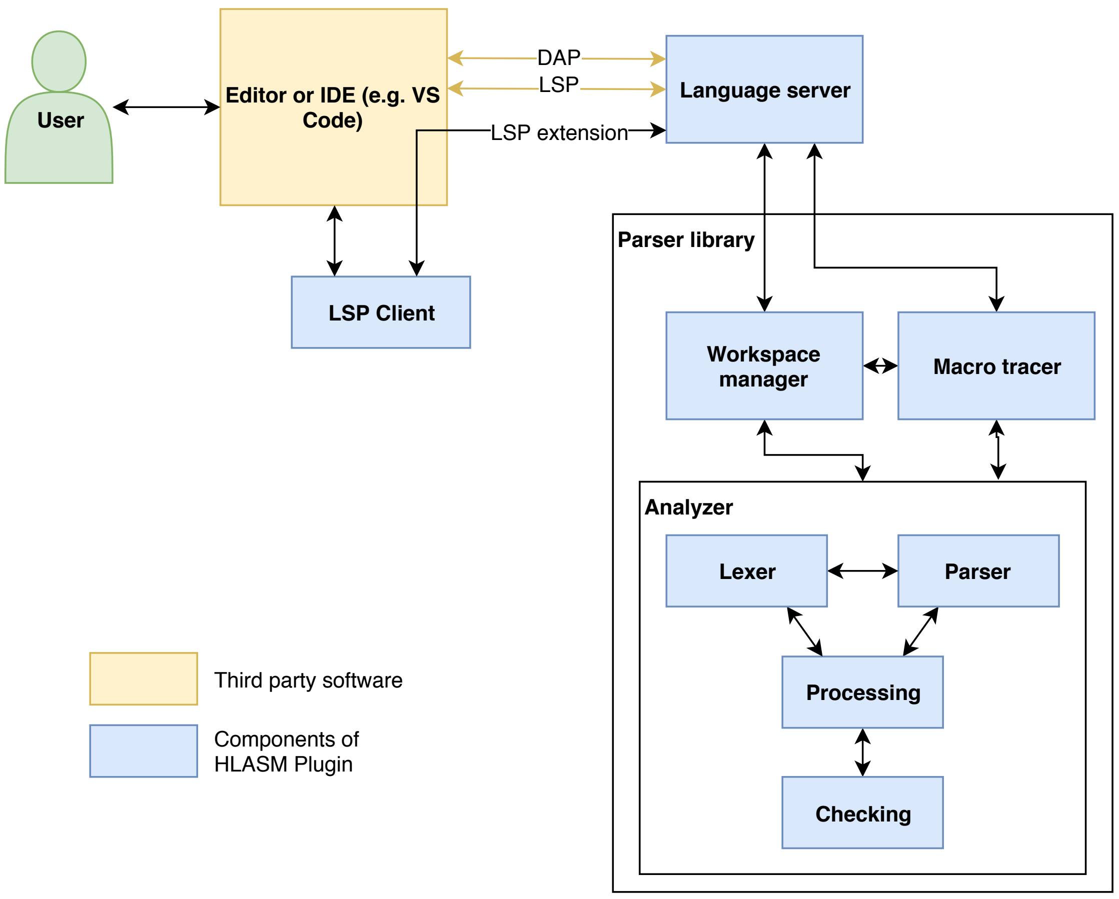

Figure 4.2: The architecture of HLASM Plugin

4.2 Parser library

Parser library is the core of the project — it encapsulates all parsing capabilities. It keeps track of opened files in the editor and provides information about them. It has API based on LSP — every relevant request and notification has corresponding method in parsing library. The API includes:

- Implementation of text synchronization notifications (didOpen, didChange, didClose), which inform the library about files that are currently opened in the editor and their exact contents.
- Implementation of workspace management notifications (DidChangeWorkspace-Folders): many editors have possibilities to open more workspaces in the same time, the parser library supports this too. Workspace is basically just a folder which contains related source codes. Workspaces help parser library find macro and copy files.
- A method to consume DidChangeWatchedFiles notification which makes it possible to react to workspace changes that were not made by the user in editor, but still may affect the parsing. For example when user deletes an external macro file, the parser library should react by reporting that it cannot find the macro.
- Implementation of diagnostics publishing (publishDiagnostics notification). A diagnostic is used to indicate a problem with source files, such as compiler error or warning. The parser library provides a callback to let language server know that diagnostics have changed.
- Callback for highlighting information provision.
- Implementation of language feature requests (definition, references, hover, completion), which provide information needed for proper reaction of the editor on user actions.

The parser library is further decomposed into smaller components.

4.2.1 Analyser

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

- Recognition of statements and their parts (lexing and parsing).
- Interpretation of instructions that should be executed in compile time.
- Check whether the HLASM source code is well-formed.
- Report problems with the source by producing LSP diagnostics.
- Provide highlighting and LSP information.

One HLASM file may have dependencies — other files, that define helping macros or files brought in by COPY instruction. The dependencies are only discovered during processing of files, so the analyzer needs a way to invoke a method that would find a file with specified name, parse its contents and return it as list of parsed statements.

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

The analyser is further decomposed into 4 components.

4.2.1.1 Lexer

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

- Check whether all characters ale valid in HLASM source
- Each token has location in the source text
- Ability to jump in the source file backward and forward is necessary for implementation of instructions like AGO and AIF.

4.2.1.2 Parser

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

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

4.2.1.3 Processing

Results of the parser component are further analysed in processing component. The most important features are:

- CA instructions are interpreted here: that results in modifying lexer state (moving back and forth in the input file).
- Substitution of variable symbols. After the substitution, the statement must be reparsed in the lexer and parser again.
- Interpretation of assembler instructions to evaluate ordinary symbols.
- MACRO and COPY expansion.

4.2.1.4 Checking

After a statement is fully processed, all operands of each instruction are known, it needs to be checked. There are over 2000 machine instructions with variable number of operands and various restrictions on those operands — some of them take only positive numbers, only numbers that are in specific range or take address only. Checking component takes an instruction and list of its operands and returns list of problems if form of LSP diagnostics.

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

4.2.2 Workspace manager

Workspace manager responsibility is to keep workspaces and files representation in parser library exactly the same as the user sees in the editor. Further, it starts analyser when needed, manages workspace configuration and provides external macro and copy libraries to analyser.

4.3 VS code extension

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

- It starts LSP client that comes with VS Code, HLASM language server and creates connection between them.
- It implements server-side highlighting, which extends the LSP protocol.
- It improves user experience regarding continuations and fixed length line source codes.

4.4 Macro tracer

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

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

- Launch, continue, next, stepIn and disconnect requests allow user to control the flow of the compilation.
- SetBreakpoints get 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 current processing stack (stack of nested macros and copy instructions), available variable symbols and their values.
- Stopped, exited and terminated events to let the DAP client know about state of traced source code.

Macro traces communicates with the workspace manager to retrieve contents of traced files. Further, it starts analysing the source file in a separate thread and get callbacks from the analyser before a statement is processed. In the callback, tracer puts the thread to sleep and wait for user interaction. During this time, it is possible retrieve all variable and stack information from the processing to show it to the user.

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

5. Technologies

mirko: soupis konkretnich technologii a verzi antlr cmake jenkins json lib boost asio? docker

vscode theia che produkcne zdrojaky poskytnute broadcom google test

–jenkins sa opytat ako s tym ze to nie je nase

jazyky typescript c++ cmake

tohle patri do Architecture, pripadne to prejmenujte na 'Implementation details' nebo tak cosi.

6. Project execution

In the following chapter is represented execution of the High Level Assembler Plugin software project. We analyze the problem difficulty, break it into tasks and estimate time requirements of particular tasks. We further describe the team and work organization.

6.1 Tasks

We analyzed the problem and split it into several tasks. At the time of writing this document implementation is already in the 24. week of project schedule. By now there is a working prototype. Therefore, some of the presented tasks are specified.

Tasks were assigned to individual team members during stand ups. The tasks and their assignment (team member name initials in the parentheses following task name) is presented in the Gantt diagram(s) (6.1, 6.2, 6.3). Project implementation is planned to be done within nine months.

6.2 Collaboration

The team consists of five members. Collaboration within the team is essential for successful completion of the project. We use a variety of means to achieve this.

Our team works with agile software development. To aid this we use visual process management system Kanban. The team meets every week together with our supervisor at stand ups. The team discusses the current status of particular tasks with their owners, review progress and plan work for next week.

For communication between team members is used online tool Slack.

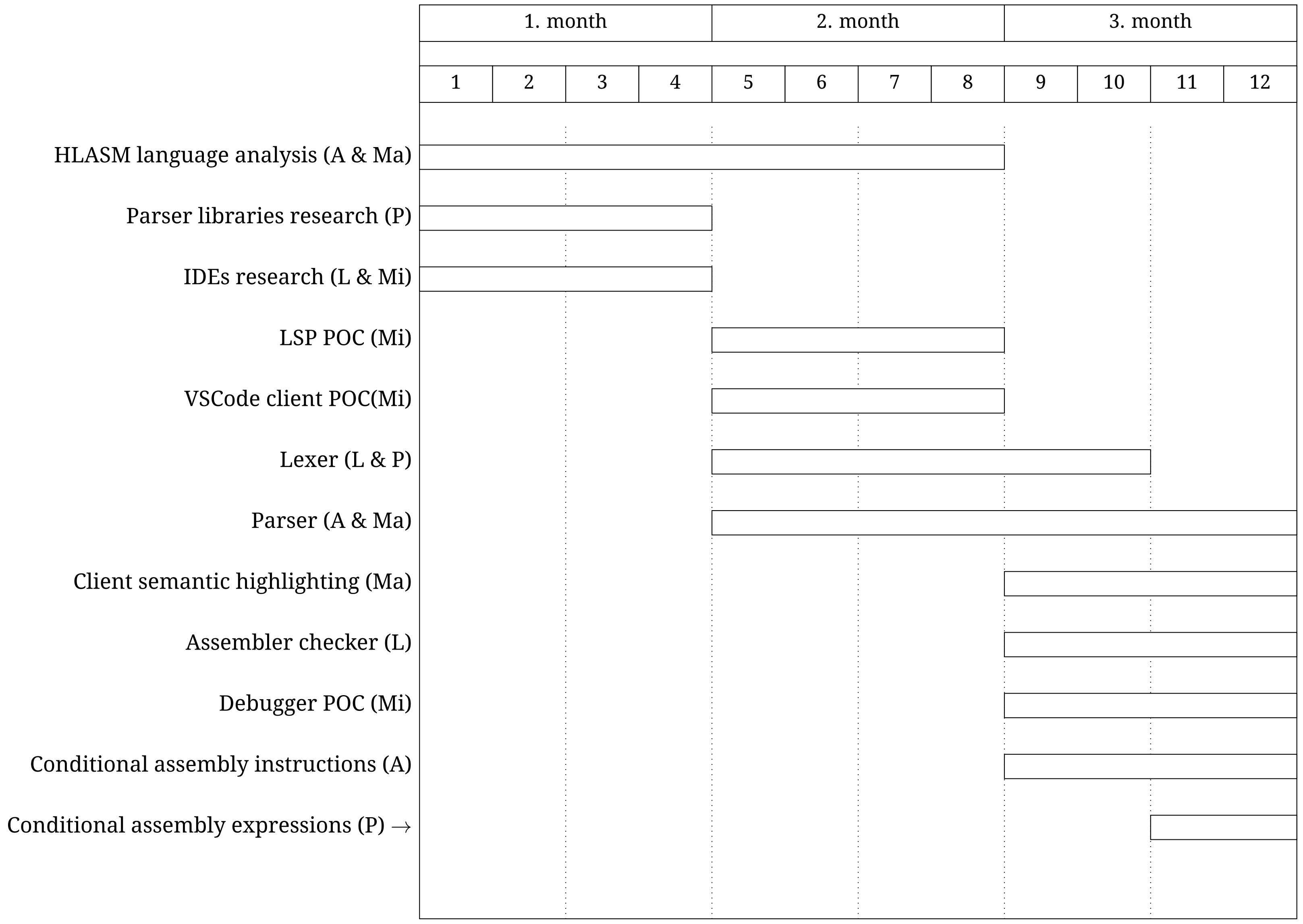

Figure 6.1: Tasks for months 1 – 3

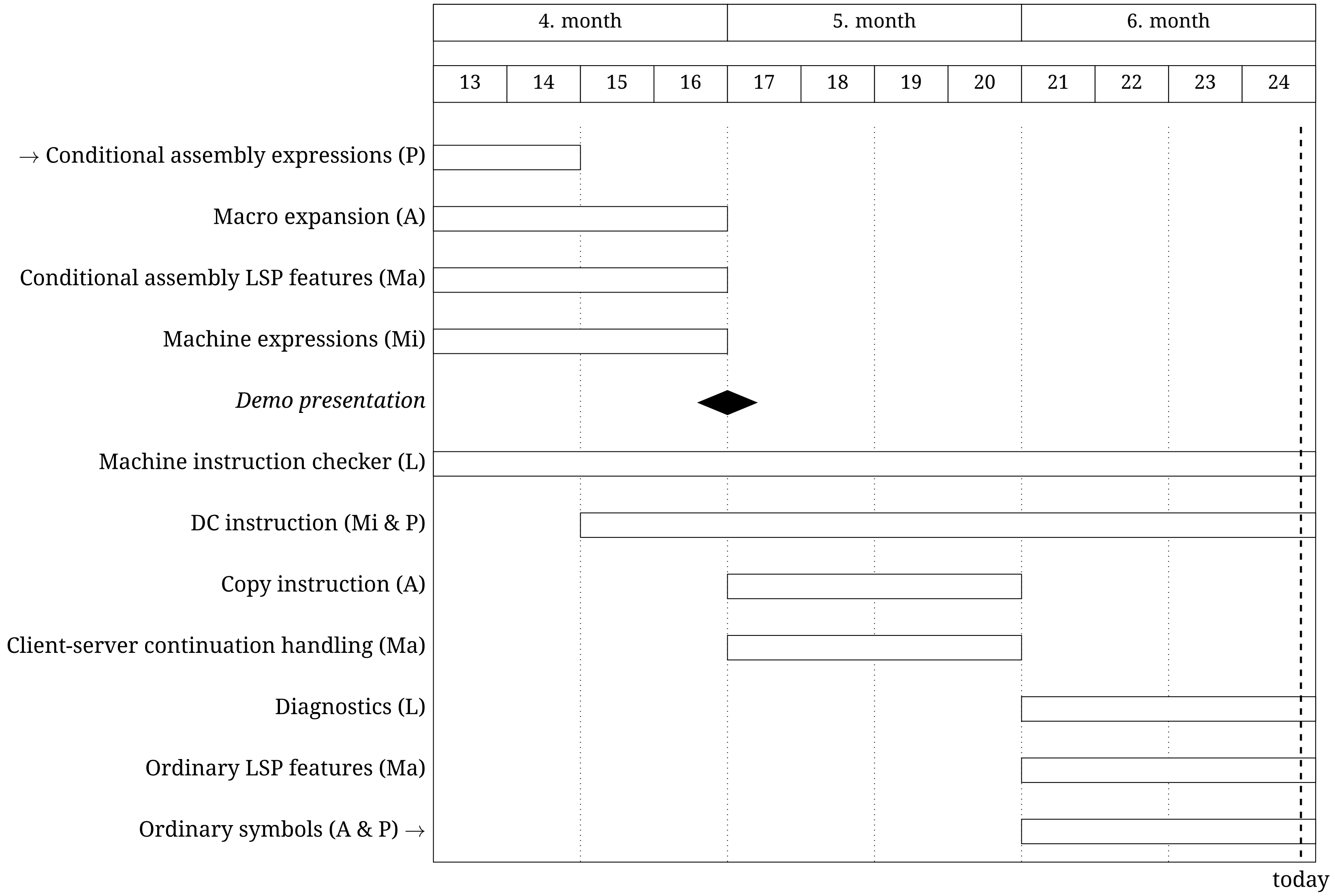

Figure 6.2: Tasks for months 4 – 6

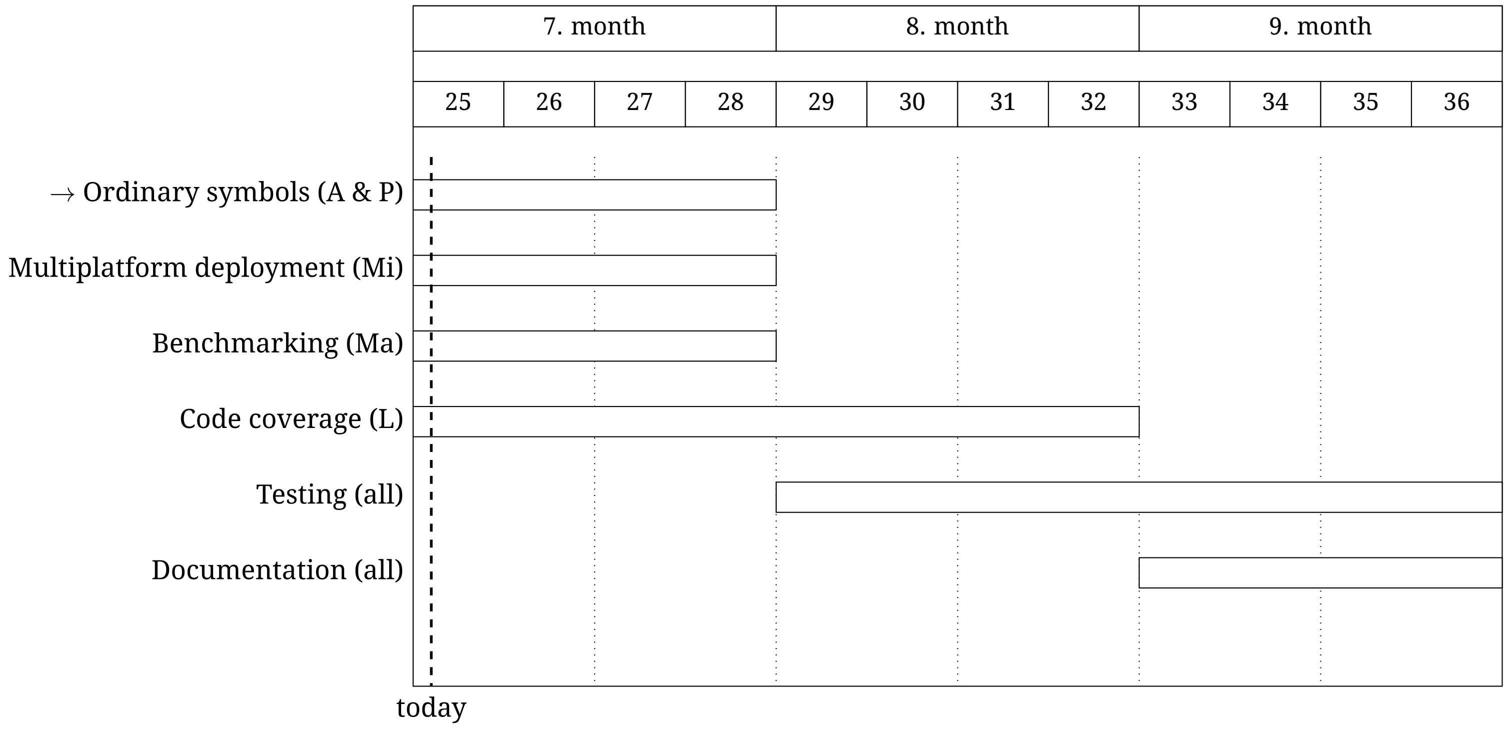

Figure 6.3: Tasks for months 7 – 9

mirko:
milestony
gantt
prirazeni lidi k projektum
udelejte si cas na psani dokumentace
je fajn mit contingency plan, co delat kdyz se to dojebe nebo ltery ficury jsou jak prioritni

je fajn vsechno tohle podeprit tim ze mate prototyp, a jak se na nej bude navazovat. Rozhodne do specifikace uz nemuzete psat ze budete volit parser a ide, protoze to tady ma bejt specifikovany.