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

Assembly languages consist of solely 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 chapter.

2.1 Syntax

Due to historical reasons, HLASM syntax differs greatly to the syntax of modern programming languages. It mostly uses syntax common to regular assemblers, which 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 is represented by a sequence of *statements*. A statement consists of four fields. These are:

- **Name field** — Serves as a place for named constants that are to be used in code. This field is optional, but, when present, it must start at the begin column of a line.
- **Operation field** — The only mandatory field representing the instruction that is executed. 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 operation 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, the operation field.

This is an example of a basic statement containing all fields.

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

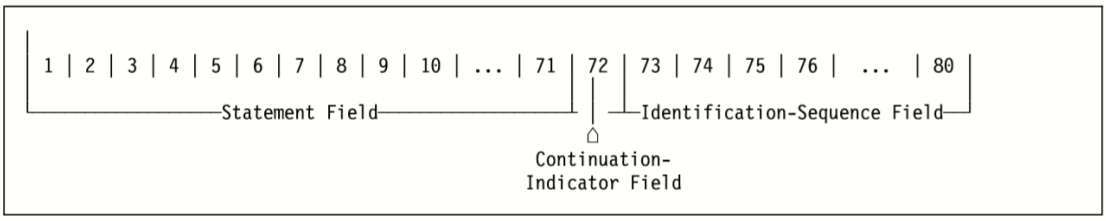

Figure 2.1: Description of line columns (source: HLASM Language Reference https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3sc264940/-$file/asmr1023.pdf).

2.1.2 Continuation

Individual statements sometimes contain more than 80 characters, which contradicts to line length limitations. Therefore, a special handling called *continuation* is introduced..

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.

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

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

They all serve a different purpose.

Begin column defines either the start of a statement, or the beginning of the name field.

End column determines the end of the statement. Anything written to its right does not count as content of the statement, and is rather used as a place for the line sequence number (see 2.1).

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*.

Below 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
```

There also exist instructions that support a so called *extended format* of the operands. It allows the presence of a continuation character 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 its syntax, this section prepares the reader to better understand the assembly process hidden behind HLASM.

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

2.2.1 Conditional assembly

The *conditional assembly* process can be compared to a C++ text preprocessor. However, in HLASM, the preprocessing is more complex, so it has obtained the term *code generation*. It works with *variable symbols*, *conditional assembly (CA) instructions* and *macros*.

2.2.1.1 Variable symbols

Variable symbols serve as points of substitution or information holders.

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

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

2.2.1.2 CA instructions

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

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

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

2.2.1.3 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 even contain a sequence of instructions generating another macro definition.

With the help of variable symbols, HLASM macros have the power to create custom, task specific macros.

2.2.2 Ordinary assembly

Ordinary assembly is a term for assembly other that conditional. It includes assembly of both machine and assembler instructions.

Machine instructions and their operands are translated to a sequence of bytes and written to the executable program. In contrast to basic assemblers, HLASM allows expressions to be passed as operands of these instructions. These expressions are capable of address arithmetics, and can also contain defined constants.

Assembler instructions, on the other hand, are used to alter the behavior of the assembler. Therefore, they are not assembled into the executable program.

2.2.2.1 Assembler instructions

A few specific instructions exist that alter assembler's behavior. Let us name a few of them:

- **ICTL** — Changes values of the previously described line columns (i.e. begin column may begin 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 an integer or a relative address value. These constants can be accessed by *conditional assembly*, hence alter it in custom manner.
- **COPY** — Copies a 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 as the start of relative addressing.

Below is an example of a 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
```

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

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

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

Line 08 establishes a start of an executable section TEST.

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

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

In line 15, the constant LEN is assigned the 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 an empty assembler action and line 19 ends the assembling of the program.

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

-JNI? asi by som nespominal
mirko:
a je fajn rozepsat vsechny API a takovy veci co sou po ceste
–velky graf vsetkych komponent –ku kazdemu odstavcek

4.1 Parser library

4.1.1 Workspace manager

4.1.2 Analyzer

4.1.2.1 Lexer

4.1.2.2 Parser

4.1.2.3 Processing

4.1.2.4 Checking

4.1.3 Debugger

4.2 Language server

4.3 VS code client

5. Technologies

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

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

vscode theia che produkcne zdrojaky poskytnute broadcom google test

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

jazyky typescript c++ cmake

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 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.

6.2 Milestones and work packages

We analyzed the problem and split it into several milestones and work packages. At the time of writing this document milestone Preview. By now there is a working prototype. Therefore, some of the presented work packages 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 diagrams in figs. 6.1 to 6.3. Project implementation is planned to be done within nine months.

(M1) Research and analysis *(month 2)*

- **(WP1)** HLASM language analysis *(Adam, Marcel)*
- **(WP2)** Parser libraries research *(Peter)*
- **(WP3)** IDEs research *(Michal, Lucia)*

(M2) Parser prototype *(month 3)*

- **(WP4)** Lexer *(Lucia, Peter)*
- **(WP5)** Parser *(Adam, Marcel)*

(M3) IDE integration prototype *(month 3)*

- **(WP6)** LSP POC *(Michal)*

(WP7) VSCode client POC *(Michal)*

(WP8) Debugger POC *(Michal)*

(M4) Preview *(month 4)*

(WP9) Client semantic highlighting *(Marcel)*

(WP10) Assembler checker *(Lucia)*

(WP11) Conditional assembly instructions *(Adam)*

(WP12) Conditional assembly expressions *(Peter)*

(WP13) Macro expansion *(Adam)*

(WP14) Conditional assembly LSP features *(Marcel)*

(WP15) Machine expressions *(Michal)*

(M5) Detailed specification *(month 6)*

(WP16) Detailed specification *(all)*

(M6) Final version skeleton *(month 7)*

(WP17) Machine instruction checker *(Lucia)*

(WP18) DC instruction *(Michal, Peter)*

(WP19) Copy instruction *(Adam)*

(WP20) Client-server continuation handling *(Marcel)*

(WP21) Diagnostics *(Lucia)*

(WP22) Ordinary LSP features *(Marcel)*

(WP23) Ordinary symbols *(Adam, Peter)*

(M7) Feature integration *(month 7)*

(WP24) Feature integration *(all)*

(M8) Feature testing *(month 9)*

(WP25) Testing *(all)*

(WP26) Multiplatform deployment *(Michal)*

(WP27) Code coverage *(Lucia)*

(M9) Documentation *(month 9)*

(WP28) Documnetation *(all)*

(M10) Final presenation *(month 9)*

(WP29) Final presenation *(all)*

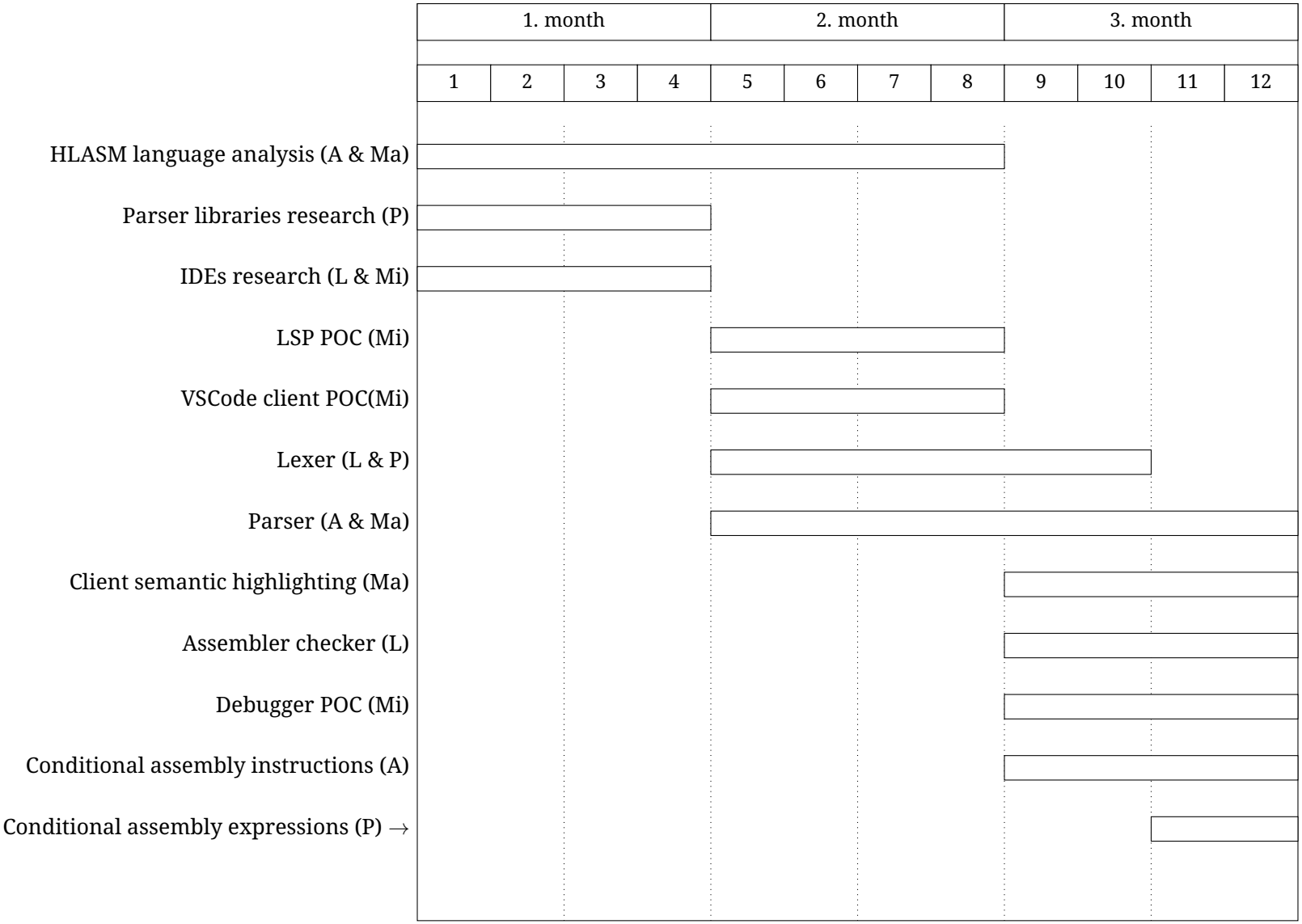

Figure 6.1: Tasks for months 1 – 3

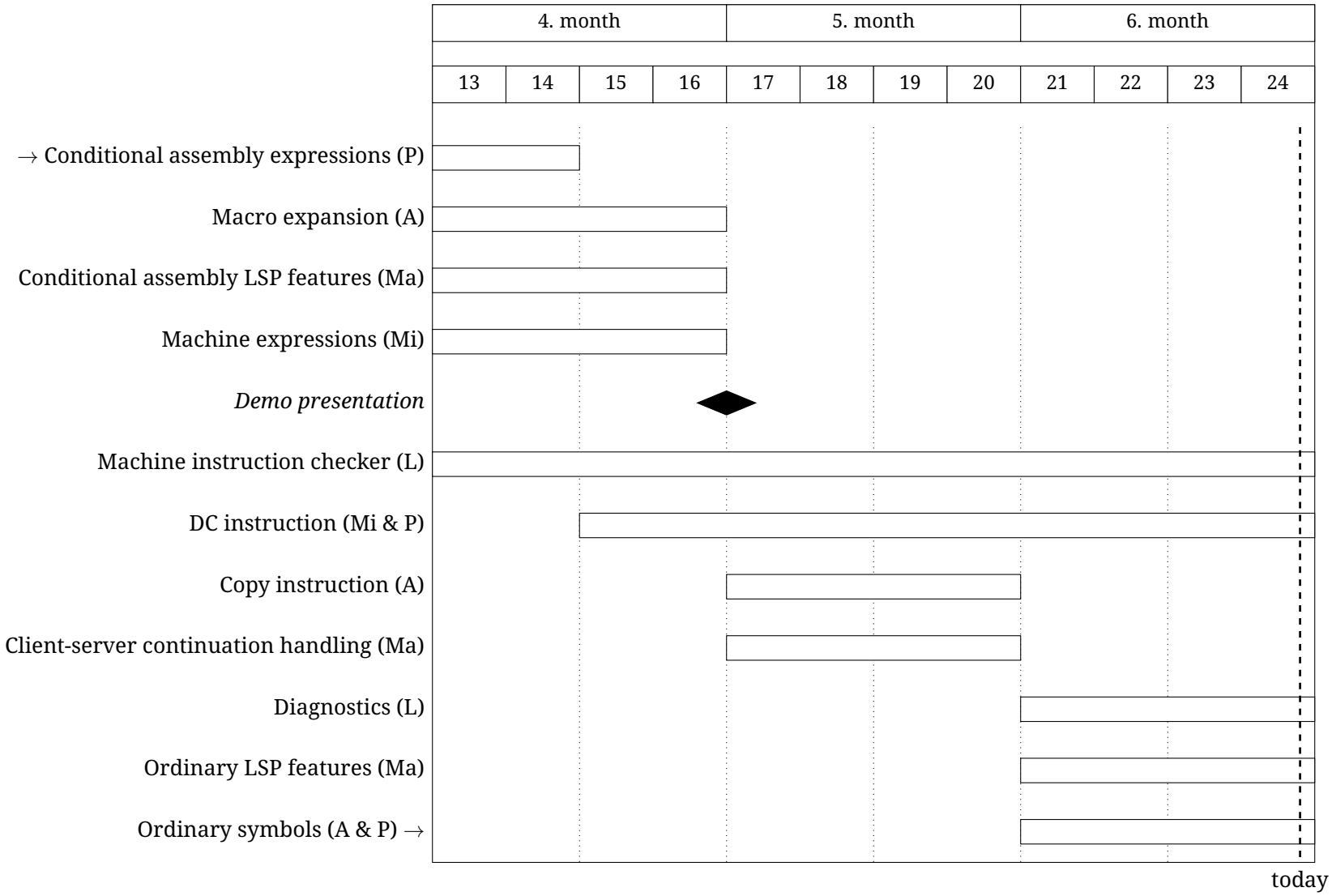

Figure 6.2: Tasks for months 4 – 6

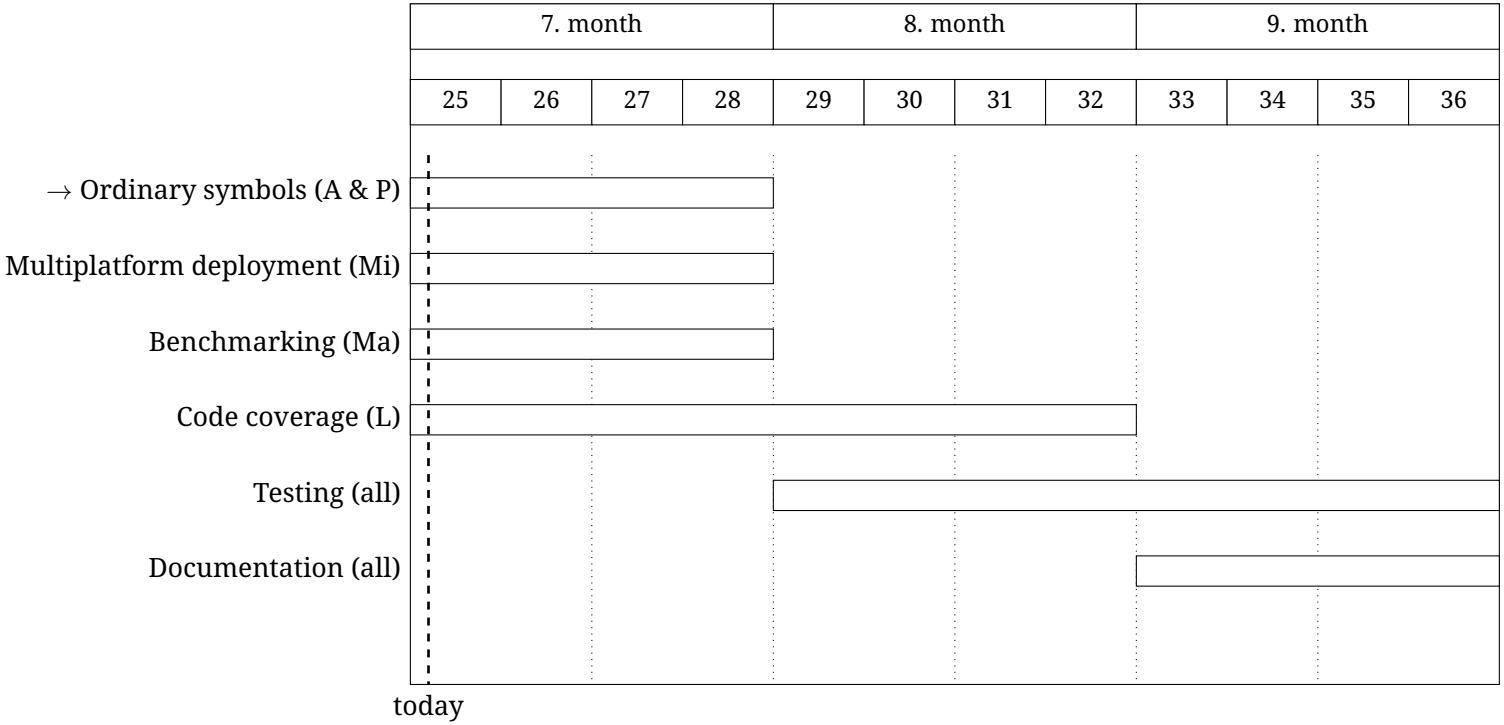

Figure 6.3: Tasks for months 7 – 9

mirko:
milestony
gantt (pokusy v fig. 6.4)
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.

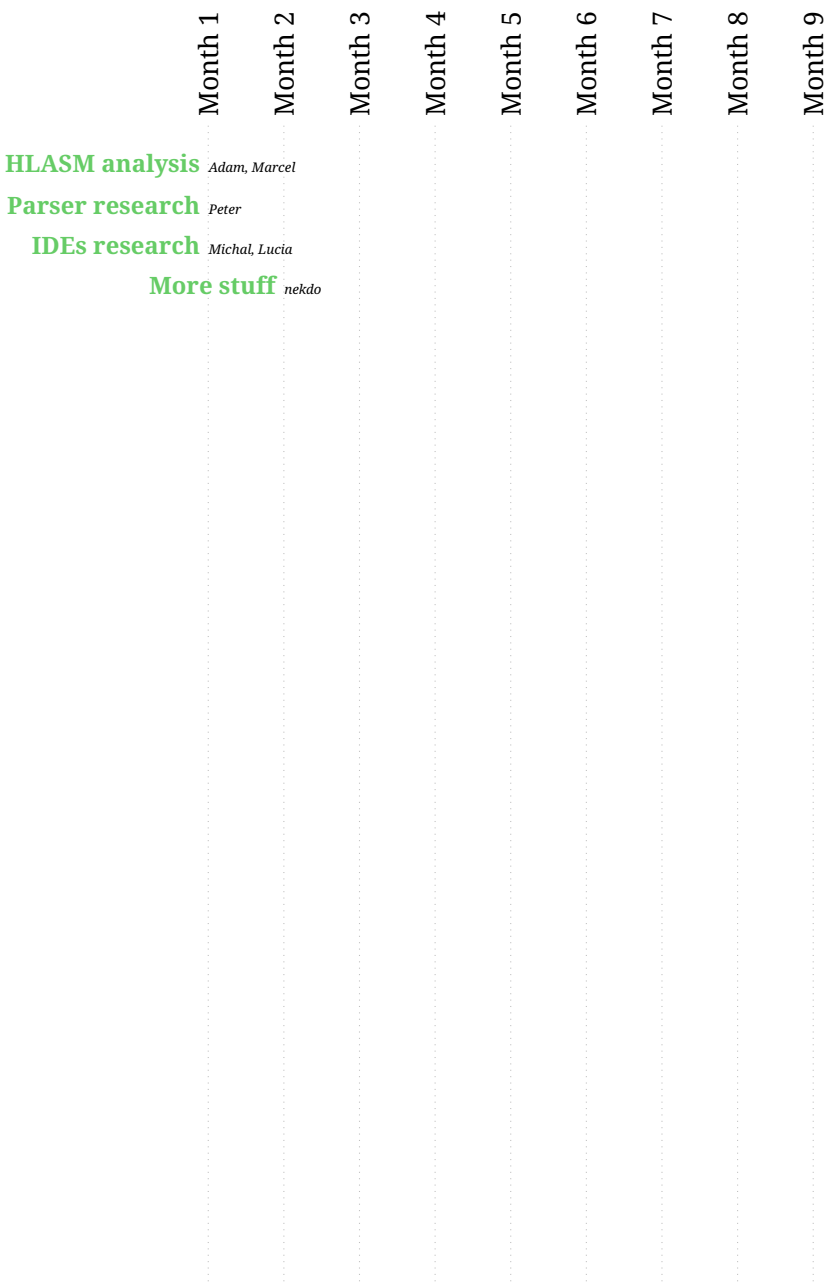

Figure 6.4: Gantt attempts.