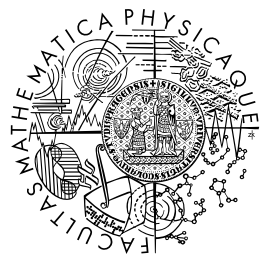

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

High Level Assembler Plugin

Detailed Specification

Authors: Bc. Michal Bali, Bc. Marcel Hruška, Bc. Peter Polák, Bc. Adam Šmelko, Bc. Lucia Tódová

Supervisor: RNDr. Miroslav Kratochvíl

Consultant: Ing. Slavomír Kučera

Prague 2019

Contents

Introduction **3**

1 IBM High Level Assembler Language overview **5**
1.1 Syntax 5
1.1.1 Statement 5
1.1.2 Continuation 5
1.2 Semantics 6
1.2.1 Conditional assembly 6
1.2.2 Ordinary assembly 6

2 Requirements **7**
2.1 Language features 7
2.2 LSP features 7

3 Architecture **9**
3.1 Language server 9
3.2 Parser library 11
3.2.1 Workspace manager 11
3.2.2 Analyser 11
3.2.3 Debugger 11
3.3 VS code client 11

4 Technologies **13**

5 Project execution **15**

Introduction

Related Work

1. IBM High Level Assembler Language 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.

1.1 Syntax

HLASM has somehow complicated syntax.

1.1.1 Statement

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

- **Name field** — Serves as place for named constants that are to be used in code.
- **Operation field** — Instruction that is executed.
- **Operands field** — Field for instruction operands separated by comma.
- **Remark field** — Serves as line commentary.

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

1.1.2 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**.

For indication that statement continues on the next line a character other than space is placed in **continuation column** (default is 72). Then the remainder of the statement must start on **continue column** (default is 15) to finally create a well formed statement (see 1.1).

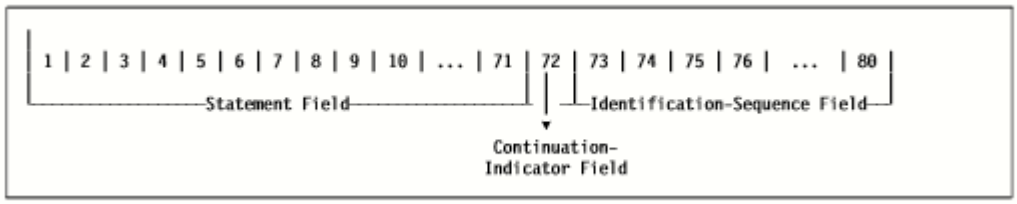

Figure 1.1: Description of line columns.

1.2 Semantics

1.2.1 Conditional assembly

1.2.2 Ordinary assembly

2. 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

2.1 Language features

-zoznam veci jazyka co podporujeme

2.2 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??

3. 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*. The client runs as an extension of development tool. 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.

This chapter presents decomposition of the project into smaller components and describes their relations. The overall architecture is pictured in Figure 3.1.

3.1 Language server

The responsibility of the Language server component is to implement LSP. The features include: and pass all the calls to parser library. The issues that it addresses:

- 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. For example 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/

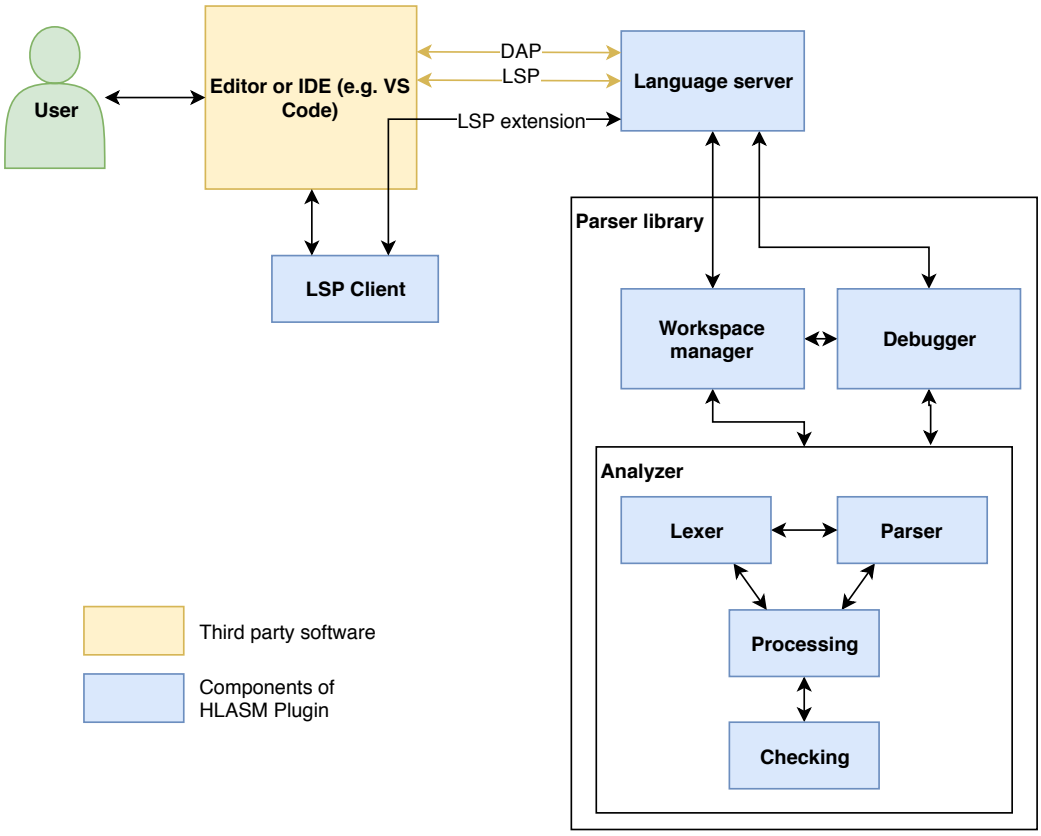

Figure 3.1: The architecture of HLASM Plugin

3.2 Parser library

3.2.1 Workspace manager

3.2.2 Analyser

Lexer

Parser

Processing

Checking

3.2.3 Debugger

3.3 VS code client

mirko:

a je fajn rozepsat vsechny API a takovy veci co sou po ceste

–velky graf vsetkych komponent –ku kazdemu odstavcek

4. 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

5. Project execution

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

1. mesiac research jazyk 8t research zvolit parser 4t research zvolit ide 4t
2. mesiac implementacia LSP POC VSCode klient POC lexer 6 parser 8t cmake 1t
3. mesiac do klienta semanticky highlighting assembler checker conditional assembly instructions expressions debugger POC
4. mesiac CA LSP features machine instruction checker 12t macro expansion
5. copy 4t machine expression 4t client-server continuation handling
6. DC ordinary 8t diagnostikz
7. ORDINARY LSP features code coverage 8t
8. benchmark testing 8t
9. dokumentacia