




Moderní trendy v korporátních informačních technologiích



Moderní trendy v korporátních informačních technologiích

- Predstavíme si zľahka HLASM a priblížime problematiku jeho spracovania
 - Hlavnou náplňou predmetu je prispieť do projektu HLASM Language Support - pridáva podporu pre HLASM do Visual Studio Code
 - Na začiatku semestra zopár prednášok na vysvetlenie základov, potom samostatná práca na vybratej úlohe s konzultáciami.
-
- Stranka predmetu: <https://github.com/michalbali256/uvod-do-smf>
 - Kontakt: michalbali32@gmail.com



Dnešný program

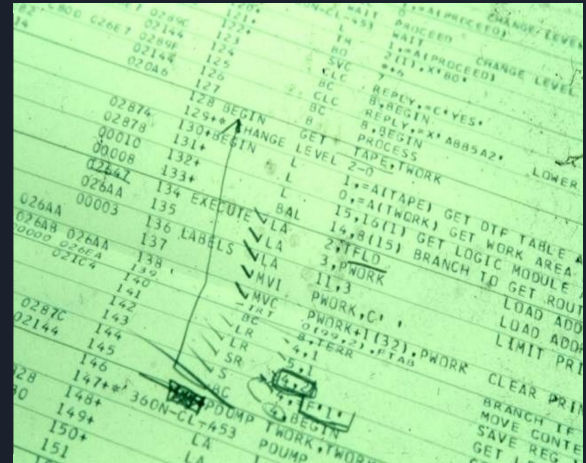
- Syntax HLASM a ako sa kompiluje.
- Čo je to HLASM Language support
- Témy úloh na semester

A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

High Level Assembler Language support

IBM High Level Assembler

- Current version released in 1992
- Origins in 1960s
- Still developed, maintained and used
- Multiple products maintained by Broadcom, hundreds of megabytes of code



Mainframe in 2020

- Old green-screen terminal
- Tedious workflow
 - Change the source code
 - Run build job
 - Check the listing
 - >30 seconds to notice a typo...

```
File Edit Edit Settings Menu Utilities Compilers Test Help
VIEW RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04 Columns 00001 00080
Command ==> Scroll ==> CSR
***** Top of Data *****
000001 *****
000002 * SIMPLE DUMMY EXIT FOR HLASM
000003 *****
000004 *
000005 * MAIN PROGRAM STARTS HERE
000006 *
000007 LARLLOAD CSECT
000008 LARLLOAD AMODE 31
000009 LARLLOAD RMODE 24
000010 * USUAL PROGRAM SETUP
000011 STM 14,12,12(13)
000012 BALR 12,0 GET THE CURRENT ADDRESS
000013 USING *,12 USE 12 AS THE BASE REGISTER
000014 L 1,=F'12'
000015 LMRET LM 14,12,12(13)
000016 XR 15,15
000017 BR 14
000018 * *****
000019 * END OF PROGRAM
000020 * *****
000021 END
***** Bottom of Data *****
```



HLASM Language Support

- Open Source plugin for modern IDEs, such as Visual Studio Code
- Provides:
 - Semantic highlighting
 - Modern IDE features, e.g. go to definition or hover
 - Validation of all machine instructions
 - Interpretation of a large subset of code-generating instructions
 - Tracing of macro expansions similar to debugging procedure
- Compliant with LSP and DAP specifications





Basic use cases

“Am I using this LR instruction correctly?”

Mainframe workflow:

- Find the instruction in the HLASM documentation OR
- Guess and try to compile it

HLASM Plugin workflow:

- In case you are not, the error message with the explanation will appear
- Autocomplete for instructions with information about their format

“Where was this VAR symbol defined?”

Mainframe workflow:

- List through tens of source files manually

HLASM Plugin workflow:

- The go to definition feature on the VAR symbol will take you to the definition



High level assembler

There are 3 types of instructions:

- Machine instructions
- Assembler instructions
(static variables, object layout, modifications of assembler program state)
- Conditional Assembly instructions
(basically a Turing-complete macro system)

Object code layout interpretation

Source file

1	PART1	LOCTR	
2			
3	L1	ST	13,4(,1)
4			
5	PART2	LOCTR	
6			
7		ORG	*,16
8			
9	L2	ST	1,8(,13)
10	L3	LR	13,1
11			
12	PART1	LOCTR	
13			
14	L4	LR	1,1

20
Relocatable Symbol
L: 2
T: 1

8



Object file

0	L1	ST	13,4(,1)
4			
4	L4	LR	1,1
6			
6	_undefined_		
16			
16	L2	ST	1,8(,13)
20			
20	L3	LR	13,1
22			

Conditional assembly instructions

- Turing-complete compile time metalanguage

```
*****.FACTORIAL.*****
*The following recursive macro computes factorial of its parameter
*during compile time. The result will be in variable symbol RES.
.....MACRO
.....FACT &A
.....
.....GBLA RES We create global variable symbol RES
*It is created with default value 0, so initialize it to 1 if this is
*the topmost call of this macro.
.....AIF (&SYSNEXT NE 1).INITEND
&RES .....SETA 1
.INITEND .....AIF (&A EQ 0).SKIP .....Nothing to do if computing fact of 0.
&RES .....SETA &RES*&A .....Do the actual multiplication
*The 'NEXT' variable is local to the scope of current macro execution.
&NEXT .....SETA &A-1 .....
.....FACT &NEXT .....Do the factorial of a-1
.SKIP .....ANOP
.....MEND
.....
.....GBLA RES We have to unhide the global variable RES, .....X
.....otherwise, it would not be visible to this scope.
.....FACT 5 Now we 'call' the macro, and it will assign 5! to X
.....&RES
```



HLASM Language Support features

- Diagnostics - extension shows an error in the IDE, when the source code would not compile on mainframe - includes more or less complete list of HLASM instructions and interpretation of most important assembler and CA instructions
- Go to definition and Go to references for ordinary, variable and sequence symbols, macro definitions, COPY files
- Context-aware highlighting
- Completion for instructions, macro names and variable symbols
- Hover over symbols in source codes to show additional information that the parser has available

Macro Tracer

- Debugger for code generation
- Traces compile-time variables and macro expansions
- Starts an analyzer in a new thread
- Listens for callbacks and stops the thread using conditional variable when a breakpoint is hit
- While the thread is stopped it is possible to inspect parsing context and show it to the user

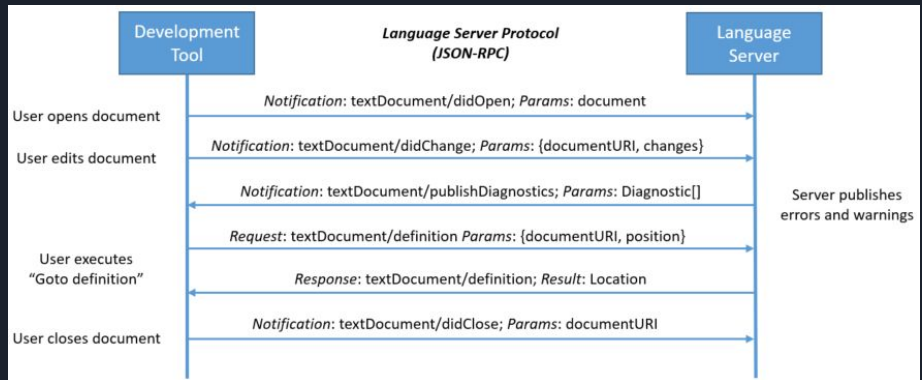
The screenshot shows the Visual Studio Macro Tracer interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The main window is divided into several panes:

- VARIABLES**: This pane is expanded, showing **Globals** and **Locals**.
 - Globals**:
 - SYTIME: 17:52
 - SYSDATC: 20200522
 - SYSDATE: 05/22/20
 - SYSPPARM
 - SYSOPT_RENT: FALSE
 - Locals**:
 - VAR: 20
 - > SYSLIST: (,20)
 - SYSECT
 - SYSLOC
 - SYSNDX: 1
 - SYSSTYP: CSECT
 - SYSNEST: 2
 - > SYSMAC: (MAC2,MAC,OPEN CODE)
 - > Ordinary symbols
- WATCH**: This pane is currently empty.
- CALL STACK**: This pane shows the current call stack, which is paused on entry.

		PAUSED ON ENTRY
MACRO	mac2	3:1
MACRO	mac.asm	4:1
OPENCODE	test	6:1
- DEBUG CONSOLE**: This pane is currently empty.

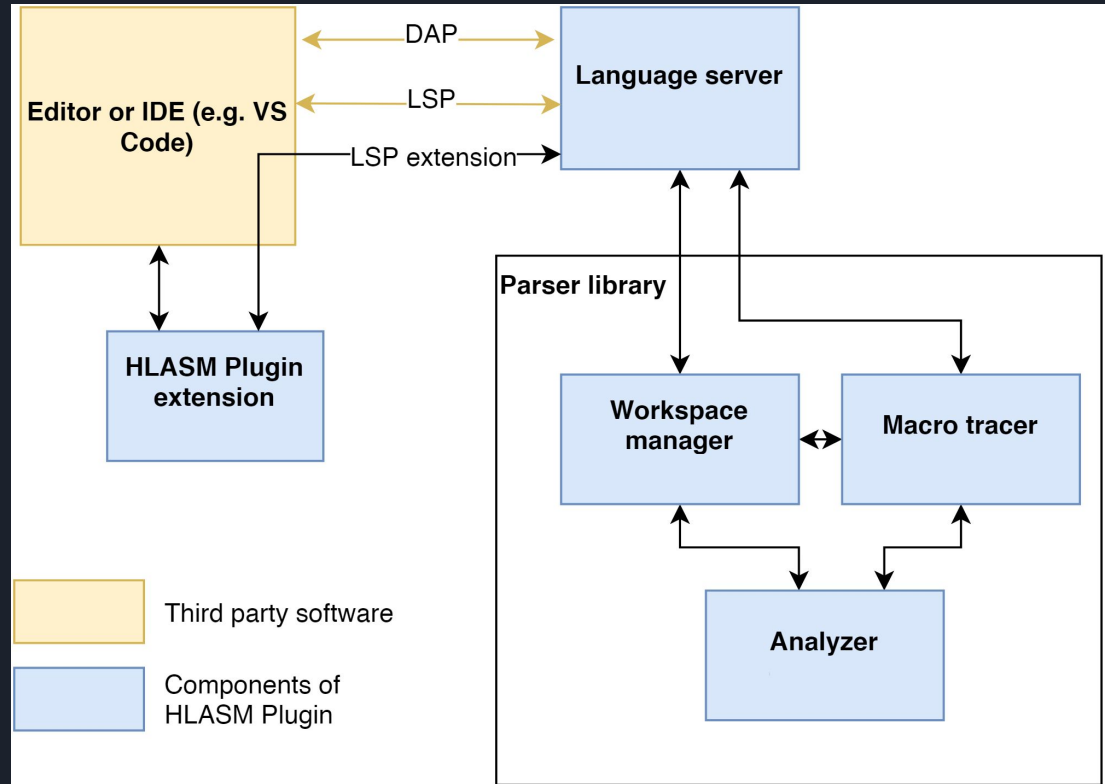
Language Server Protocol

- The Language Server Protocol (LSP) defines the protocol used between an editor or IDE and a language server that provides language features like auto complete, go to definition, find all references etc.
- Solves the N:M problem
- Based on JSON RPC



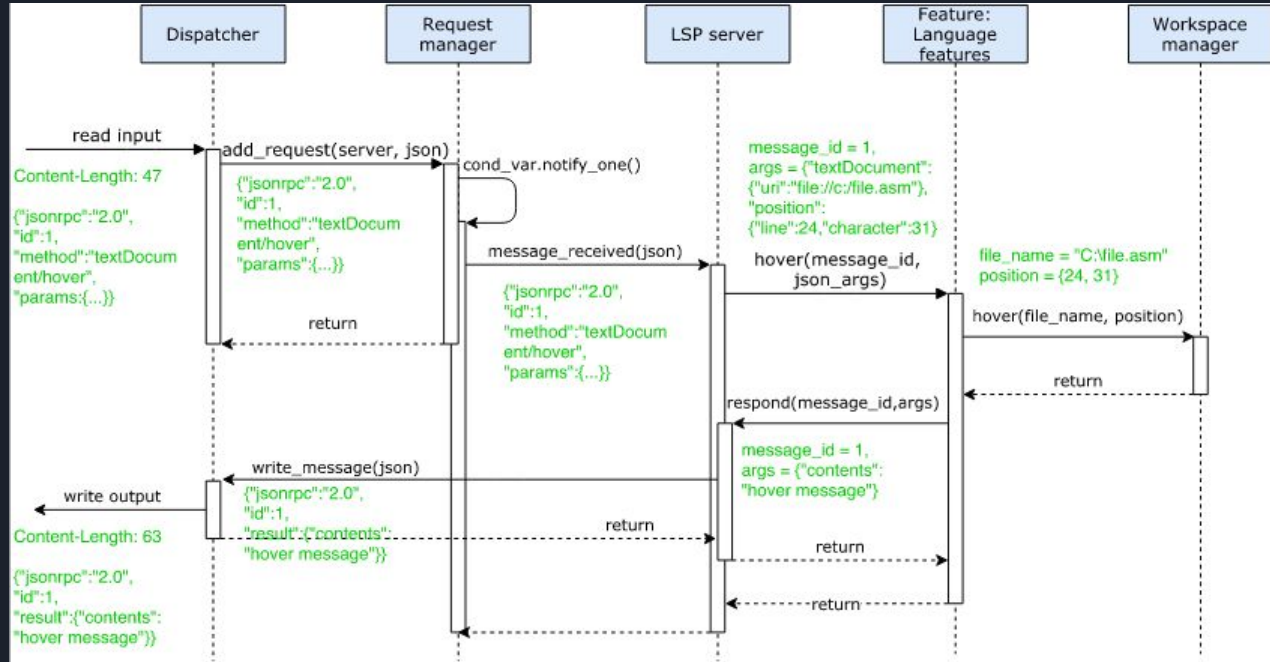
Architecture Overview

- 3 main components
 - Language server (C++)
 - Parser library (C++)
 - Workspace Manager
 - Macro Tracer
 - Analyzer
 - VSCode Extension (TypeScript)



Language server & Workspace Manager

- These components expose the analyzer to the user
- Interface compliant with the LSP and DAP specifications
- Manage input files and workspaces
- Presentation of parsing results to the user
- Uses Analyzer for processing and sends back the results

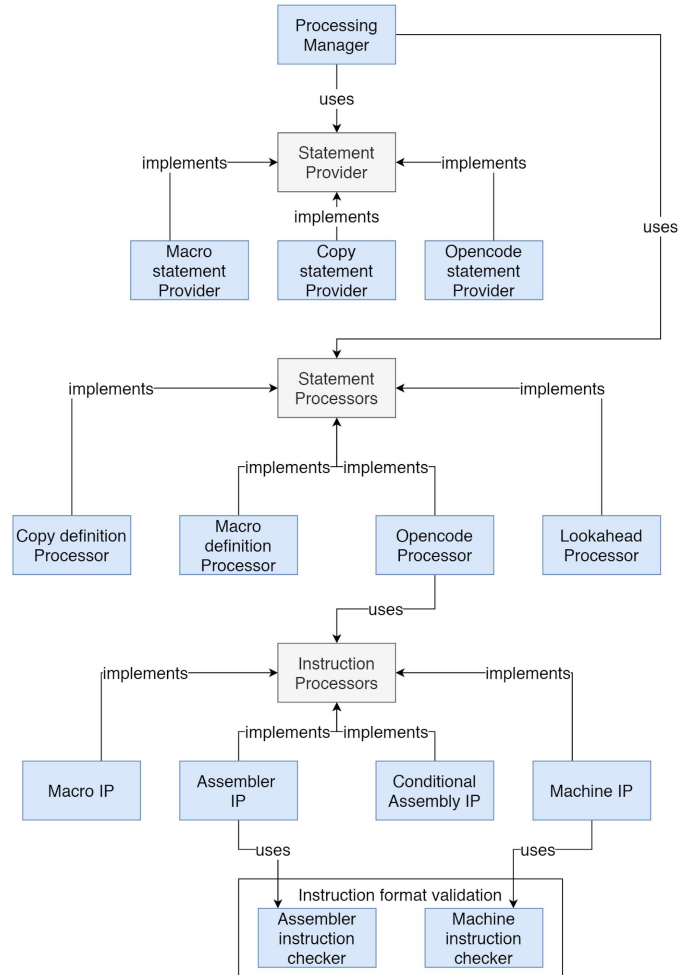


Code analyzer

- Processes the contents of HLASM source code
- Different processors for different situations
- Validates machine instructions
- Interprets assembler and conditional assembly instructions

```

1335 add_machine_instr(result, "LEXIK", mach_format::RRF_b, { reg_4_U, reg_4_U, reg_4_U }, 32, 1512);
1336 add_machine_instr(result, "LTDTR", mach_format::RRE, { reg_4_U, reg_4_U }, 32, 1513);
1337 add_machine_instr(result, "LTXTR", mach_format::RRE, { reg_4_U, reg_4_U }, 32, 1513);
1338 add_machine_instr(result, "FIDTR", mach_format::RRF_e, { reg_4_U, mask_4_U, reg_4_U, mask_4_U }, 32, 1514);
1339 add_machine_instr(result, "FIXTR", mach_format::RRF_e, { reg_4_U, mask_4_U, reg_4_U, mask_4_U }, 32, 1514);
1340 add_machine_instr(result, "LDETR", mach_format::RRF_d, { reg_4_U, reg_4_U, mask_4_U }, 32, 1517);
1341 add_machine_instr(result, "LXDTR", mach_format::RRF_d, { reg_4_U, reg_4_U, mask_4_U }, 32, 1517);
1342 add_machine_instr(result, "LEDTR", mach_format::RRF_e, { reg_4_U, mask_4_U, reg_4_U, mask_4_U }, 32, 1518);
1343 add_machine_instr(result, "LDXTR", mach_format::RRF_e, { reg_4_U, mask_4_U, reg_4_U, mask_4_U }, 32, 1518);
1344 add_machine_instr(result, "MDTR", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U }, 32, 1519);
1345 add_machine_instr(result, "MXTR", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U }, 32, 1519);
1346 add_machine_instr(result, "MDTRA", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1520);
1347 add_machine_instr(result, "MXTRA", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1520);
1348 add_machine_instr(result, "QADTR", mach_format::RRF_b, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1521);
1349 add_machine_instr(result, "QAXTR", mach_format::RRF_b, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1521);
1350 add_machine_instr(result, "RRDTR", mach_format::RRF_b, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1524);
1351 add_machine_instr(result, "RRXTR", mach_format::RRF_b, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1524);
1352 add_machine_instr(result, "SLDT", mach_format::RXF, { reg_4_U, reg_4_U, dxb_12_4x4_U }, 48, 1526);
1353 add_machine_instr(result, "SLXT", mach_format::RXF, { reg_4_U, reg_4_U, dxb_12_4x4_U }, 48, 1526);
1354 add_machine_instr(result, "SRDT", mach_format::RXF, { reg_4_U, reg_4_U, dxb_12_4x4_U }, 48, 1526);
1355 add_machine_instr(result, "SRXT", mach_format::RXF, { reg_4_U, reg_4_U, dxb_12_4x4_U }, 48, 1526);
1356 add_machine_instr(result, "SDTR", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U }, 32, 1527);
1357 add_machine_instr(result, "SXTR", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U }, 32, 1527);
1358 add_machine_instr(result, "SDTRA", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1527);
1359 add_machine_instr(result, "SXTRA", mach_format::RRF_a, { reg_4_U, reg_4_U, reg_4_U, mask_4_U }, 32, 1527);
1360 add_machine_instr(result, "TDCET", mach_format::RVE, { reg_4_U, dxb_12_4x4_U }, 48, 1528);
    
```



VSCode Extension

- End-user component
- Standard extension for VSCode
- Semantic highlighting is an addition to the LSP




HLASM Language Support broadcommfd.hlasm-language-support

Broadcom | 7,398 | ★★★★★ | Repository | License | 0.11.0

Code completion, highlighting, browsing and validation for High Level Assembler.

Install

[Details](#) [Feature Contributions](#) [Changelog](#)

[open issues](#) 3 [chat](#) [on Slack](#)  scanned on sonarcloud

HLASM Language Support

HLASM Language Support is an extension that supports the High Level Assembler language. It provides code completion, highlighting and navigation features, shows mistakes in the source, and lets you trace how the conditional assembly is evaluated with a modern debugging experience.

This extension is a part of the [Che4z](#) open-source project.

HLASM Language Support is also part of [Code4z](#), an all-round package that offers a modern experience for mainframe application developers, including [COBOL Language Support](#), [Explorer for Endeavor](#), [Zowe Explorer](#) and [Debugger for Mainframe](#) extensions.

Getting Started

Usage

Follow these steps to open a HLASM project:

1. In menu *File* -> *Open Folder...*, select the folder with the HLASM sources.
2. Open any HLASM source file (note that HLASM does not have a standard filename extension) or create a new file.
3. If the auto-detection of HLASM language does not recognize the file, set it manually in the bottom-right corner of the VS Code window.
4. The extension is now enabled on the open file. If you have macro definitions in separate files or use the COPY instruction, you need to setup the workspace.

Setting up a multi-file project environment

HLASM COPY instruction copies the source code from various external files, as driven by HLASM evaluation. The source code interpreter in the HLASM Extension needs to be set up correctly to be able to find the same files as the HLASM assembler program.

This is done by setting up two configuration files — `proc_grps.json` and `pgm_conf.json`. The extension guides the user in their creation:

1. After opening a HLASM file for the first time, two pop-ups are displayed. Select *Create pgm_conf.json with current program* and *Create empty*



Technologies

- Crossplatform - Windows, MacOS, Linux
- Languages - C++17, Typescript
- Parser - ANTLR
- Build system - CMake
- Pipeline/continuous integration - Github Actions
- Sonarcloud



Themes for projects

- Outline
- Folding ranges
- Processor group switching
- HLASM listing highlighting

Outline

- A list of defined symbols in opened file
- Implementation of LSP request documentSymbol

```
C parse_lib_provider.h
G processor_file_impl.cpp
C processor_file_impl.h
C processor_group.h
C processor.h
G wildcard.cpp
C wildcard.h
G workspace.cpp
C workspace.h
```

OUTLINE

```
HLASMPUGIN_PARSERLIBRARY_F...
└─ {} hlasm_plugin
  └─ {} parser_library
    └─ {} workspaces
      └─ file_uri type alias
        └─ file
          └─ get_file_name() declaration
          └─ get_text() declaration
          └─ update_and_get_bad() declar...
          └─ get_lsp_editing() declaration
          └─ get_version() declaration
          └─ did_open(std::string, version...
          └─ did_change(std::string) decla...
          └─ did_change(range, std::strin...
          └─ did_close() declaration
```

TIMELINE

```
25 namespace hlasm_plugin::parser_library::workspaces {
26
27 using file_uri = std::string;
28
29 // Interface that represents both file opened in LSP
30 // as well as a file opened by parser library from the disk.
31 class file : public virtual diagnosable
32 {
33 public:
34     virtual const file_uri& get_file_name() = 0;
35     // Gets contents of file either by loading from disk or from LSP.
36     virtual const std::string& get_text() = 0;
37     // Returns whether file is bad - bad file cannot be loaded from disk.
38     // LSP files are never bad.
39     virtual bool update_and_get_bad() = 0;
40     // Returns whether file is open by LSP.
41     virtual bool get_lsp_editing() = 0;
42
43     // Gets LSP version of file.
44     virtual version_t get_version() = 0;
45
46     // LSP notifications
47     virtual void did_open(std::string new_text, version_t version) = 0;
48     virtual void did_change(std::string new_text) = 0;
49     virtual void did_change(range range, std::string new_text) = 0;
50     virtual void did_close() = 0;
51 };
52
53
54 } // namespace hlasm_plugin::parser_library::workspaces
55
56 #endif
57
```

Outline implementation in C++ extension



Outline

- For HLASM, it should list defined ordinary symbols (including location counters and *SECTs), sequence symbols, variable symbols
- Symbols defined in an external file should be enclosed under common node
- STRETCH - There are commonly used pair macros (FUNCENTRY-FUNCEND) that mark the beginning and the end of functions.

Folding ranges

- Currently, there are default VS Code folding ranges based solely on indentation.
- However, that does not work that well for HLASM, since label always starts at the first column and only then do we indent the instruction
- Comments should be ignored
- In presence of structured macros, the problem can be approached either based solely on indentation, or recognize the actual macros (configuration needed for the user to define which macros does he want to fold)
- Implementation of LSP request foldingRange

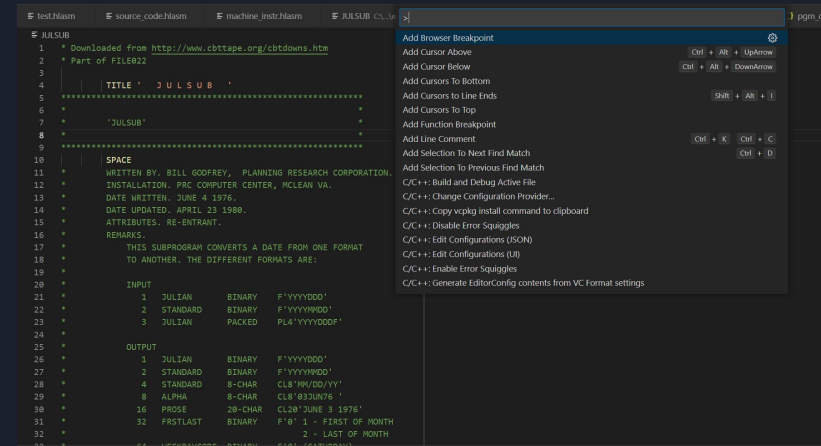
The top screenshot shows a VS Code editor with HLASM code. Folding ranges are indicated by chevrons on the left margin. The ranges are based on indentation: lines 10-15 are folded under line 9, and lines 16-18 are folded under line 13. The code is as follows:

```
9  
10 IF R15,NH,256,CHI  
11 STG R0,LAB  
12 XGR R15,R15  
13 LG R0,SS  
14 LR 1,1  
15 LR 1,1  
16 * A COMMENT  
17 IFEND ,  
18 EQU 1
```

The bottom screenshot shows the same code with a different folding configuration. The ranges are now based on macro labels: lines 10-15 are folded under line 9 because they are part of the 'IF' macro, and lines 16-18 are folded under line 13 because they are part of the 'LEN' macro. The code is the same as above.

UI element for processor group switching

- proc_grps.json define several processor groups; one processor group is essentially a list of directories where our parser should look for external macros and copy files
- pgm_conf.json defines which processor groups should be used for which source files
- For some files, there may be more processor groups - differentiating different versions of the source file.
- The task is to create a UI element that would allow to quickly choose processor group for currently opened source file (open code), i.e. modify pgm_conf.json appropriately.





HLASM listing highlighting

- A HLASM listing is a byproduct of HLASM compilation. It shows expansion of macros, variable symbols and much more additional information
- HLASM developers spend (too) much time by studying it. The first step is to have it colored
- Add a new “language” to the VS Code extension
- Write TextMate grammar

Thank you for your attention

