

## ONDERZOEKSVOORSTEL

# Statische analyse en visualisatie van control flow in HLASM-code met Mermaid.js, ter ondersteuning van beginnende mainframeontwikkelaars.

Bachelorproef, 2025-2026

Sarah Bader

E-mail: [sarah.bader@student.hogent.be](mailto:sarah.bader@student.hogent.be)

Project repo: [https://github.com/Sarah-Bader/Bachelorproef\\_26-25](https://github.com/Sarah-Bader/Bachelorproef_26-25)

Co-promotor: Bobby Tjassens Keiser (ICU IT Services, [bobby.tjassens.keiser@icu-it.nl](mailto:bobby.tjassens.keiser@icu-it.nl))

## Samenvatting

Veel applicaties op het mainframe zijn vaak afhankelijk van High-Level Assembler (HLASM), de programma's gebruiken kritische functionaliteit die vanuit een HLASM-broncode komt. Dit zorgt ervoor dat onderhoud van deze broncode van groot belang is. Wanneer deze code faalt kan de continuïteit van het systeem in gevaar komen. Momenteel wordt deze legacycode onderhouden door een beperkt aantal mainframe-specialisten, die sterk aan het afnemen zijn. Ook is deze legacycode vaak minimaal gedocumenteerd, waardoor het tijdrovend is om de functionaliteit ervan achter te halen. De mainframe-industrie heeft behoefte aan een nieuwe generatie mainframeontwikkelaars en hulpmiddelen om deze generatie te ondersteunen bij het beheersen van deze legacycode. Dit onderzoek richt zich specifiek op het ondersteunen van beginnende mainframeontwikkelaars met een tool die de control-flow van HLASM-legacycode statisch analyseert en visualiseert in Mermaid.js syntax. Concreet is de proof of concept (PoC) een op JavaScript-gebaseerde parser die een subset van de belangrijkste HLASM-instructies en labels detecteert en daarna omzet naar Mermaid.js-elementen.

De methodologie is opgedeeld in verschillende fasen, met elk een resulterend doeleind die verwerkt wordt in de volgende fase. De eerste fase bevat een literatuurstudie die dieper ingaat op het belang van de mainframe-computer, de nood aan onderhoud van HLASM-code, de moeilijkheden bij het leren van assemblytalen en de bestaande visualisatietools voor High Level Assembler. In de literatuurstudie worden ook de HLASM-instructies en labels gedefinieerd die de control flow definiëren. Na de literatuurstudie start de ontwikkeling van de proof of concept. Tijdens de ontwikkeling wordt de PoC geëvalueerd op een Agile manier, het wordt vergeleken met traditionele, handmatige code reviews. Het verwachte resultaat is dat de PoC sneller inzicht biedt in de functionaliteiten van HLASM-programma's aan beginnende mainframeontwikkelaars.

**Keuzerichting:** Mainframe Expert

**Sleutelwoorden:** Mainframe, HLASM, Control Flow Graphs, Static Analysis, Program Visualization, Legacy Systems, z/Architecture, z/OS, Mermaid.js

## Inhoudsopgave

1	Inleiding	1
2	Literatuurstudie	2
2.1	Belang van de mainframecomputer	2
2.2	Nood aan onderhoud van HLASM-code	2
2.3	Moeilijkheden voor beginnende ontwikkelaars	2
2.4	Effectiviteit van visualisatietechnieken	3
2.5	Bestaande visualisatietools	3
2.6	High Level Assembler	3
2.7	Control Flow in High Level Assembler	3
2.8	JavaScript Parsing	3
2.9	Mermaid.js en flowcharts	4
3	Methodologie	4
3.1	Fase 1: Oriëntatie en Literatuurstudie	4
3.2	Fase 2: Requirementsanalyse	4
3.3	Fase 3: Implementatie en Evaluatie	5
3.4	Fase 4: Resultaat en Verdediging	5

3.5	Tijdsplanning	5
4	Verwacht resultaat, conclusie	5
	Referenties	5

## 1. Inleiding

Tegenwoordig spelen mainframecomputers een cruciale rol in de dagelijkse activiteiten van bedrijven in de financiële sector, de gezondheidszorg, verzekeringen en tal van andere publieke en private ondernemingen. Deze bedrijven zijn afhankelijk van mainframecomputers en hebben dan ook de nood om deze zo optimaal mogelijk te onderhouden. Vermits veel ervaren mainframe-specialisten hun pensioen naderen, is er nood aan een nieuwe generatie mainframe-specialisten die het onderhoud van hun systemen kan waarborgen. Onder dit onderhoud vallen de bedrijfskritische functionaliteiten van mainframeapplicaties, die voornamelijk afhankelijk zijn van High-Level Assembler (HLASM).

Deze legacycode is vaak tientallen jaren oud en minimaal gedocumenteerd. Voor de nieuwe generatie mainframeontwikkelaars vormt dit een uitdaging om deze legacycode te begrijpen en te onderhouden. Zij vertrouwen vaak enkel op manuele code reviews.

Door middel van een geautomatiseerde tool die statische analyse van HLASM-code visualiseert in een control-flowchart, wil dit onderzoek ondersteuning bieden aan beginnende mainframeontwikkelaars, specifiek binnen organisaties die afhankelijk zijn van legacy HLASM-code.

De probleemstelling van dit onderzoek luidt als volgt:

*Hoe effectief versnellen statische analyse en Mermaid-diagrammen het leerproces van HLASM bij beginners?*

Het doel van dit onderzoek is het ontwikkelen van een proof-of-concept (PoC) die een representatieve subset van HLASM-instructies en labels statisch analyseert en deze omzet in een visueel control-flowdiagram in Mermaid.js-syntax.

Vooraleer de ontwikkeling van de PoC van start gaat, wordt een literatuurstudie uitgevoerd naar het probleemdomein en het oplossingsdomein.

Rond het probleemdomein worden volgende vragen bestudeerd:

- Wat is het belang van de mainframecomputer?
- Waarom is er nood aan het onderhouden van HLASM-code?
- Wat zijn de moeilijkheden bij het begrijpen van assemblytalen voor beginnende ontwikkelaars?

Rond het oplossings domein worden volgende vragen bestudeerd:

- Hoe helpt visualisatie bij het sneller begrijpen van programma's?
- Welke tools bestaan er al om HLASM-code te visualiseren?
- Welke HLASM-instructies zijn van noodzaak om de control-flow van de code in kaart te brengen?
- Wat is de beste manier om de kolomgebaseerde structuur van HLASM te verwerken met een JavaScript-parser?
- Hoe vertaal je de vertakkingen in de HLASM-code naar correcte Mermaid.js-diagrammen?

## 2. Literatuurstudie

### 2.1. Belang van de mainframecomputer

Vaak worden mainframes beschreven als oude technologie en wordt gedacht dat ze niet meer relevant zijn. Ook wouden bedrijven graag mainframes vervangen door andere servers. Momenteel zijn mainframes nog steeds zeer actief en staan ze in voor kritische bedrijfsactiviteiten. Tegenwoordig worden mainframes ook geïntegreerd met moderne technologieën (Sagers e.a., 2018). Het onderzoek van Sagers e.a. (2018) wordt bevestigd door recent onderzoek waarin wordt aangetoond dat organisaties niet enkel gebruikmaken van legacyapplicaties, maar ook bezig zijn met het actief ontwikkelen van nieuwe applicaties (Shaik, 2024).

Mainframes zijn dus een zeer relevante en moderne technologie, die dagelijks draait. Omdat ervaren mainframeprofessionals hun pensioen naderen is er een behoefte aan een nieuwe generatie mainframespecialisten die het onderhoud van deze systemen kunnen waarborgen (Waites & Ketterer, 2013).

### 2.2. Nood aan onderhoud van HLASM-code

Bedrijfskritische functionaliteiten van mainframeapplicaties zijn vaak afhankelijk van High Level Assembler (HLASM), waardoor onderhoud van deze legacycode van groot belang is. Als de aangeroepen HLASM-code faalt, dan faalt het volledige programma (Zaytsev, 2020). Uit een onderzoek naar softwareonderhoud blijkt dat het begrijpen van bestaande code een groot deel van de moeite neemt tijdens onderhoud. Vooral bij beginnende ontwikkelaars gaat het merendeels van hun tijd aan het begrijpen van de functionaliteit, dit is omdat zij minder ervaring hebben (Xia e.a., 2018). Dit is vooral het geval bij legacycode zoals HLASM, omdat daar de documentatie vaak beperkt is.

### 2.3. Moeilijkheden voor beginnende ontwikkelaars

Assemblytalen zijn zeer gedetailleerd, als ontwikkelaar definieer je elke kleine stap van het programma, wat ingewikkeld kan zijn voor beginners. Uit onderzoek blijkt dat naarmate de ervaring toeneemt, het gemakkelijker wordt om de functionaliteit van code te begrijpen (Feigenspan e.a., 2012).

Binnen het onderwijs hebben studenten vooral moeite met de componenten die de logische volgorde van het programma bepalen. Dit zijn de componenten die de control flow van het programma definiëren. Bij het begrijpen van de control flow maken beginners vaak fouten en er

varen ze vaak verwarring (Kawash, 2024). Hieruit kan worden geconcludeerd dat de control flow van code een moeilijkheid is onder beginnende ontwikkelaars.

## 2.4. Effectiviteit van visualisatietechnieken

Visualisatietechnieken worden vaak gebruikt om code beter te begrijpen en worden ook erkend als technieken die sneller en meer inzicht geven in codefunctionaliteiten. Visualisaties worden vaak gebruikt om de structuur van code te representeren (Storey, 2005). Bij assembly-talen kan visualisatie vooral helpen, aangezien de code vaak is opgebouwd uit zeer gedetailleerde instructies en kan worden gegroepeerd in componenten (Baldwin e.a., 2009).

## 2.5. Bestaande visualisatietools

Een bestaande tool die kan worden gebruikt voor de visualisatie van HLASM-code is FermaT. Deze tool is vooral gericht op het begrijpen en migreren van legacycode. Het is een zeer krachtige en zware tool die veel functionaliteiten heeft (Ward, 2001).

Het doel van dit onderzoek is het ontwikkelen van een lichte tool die HLASM-code omzet naar een high-level control-flowdiagram in Mermaid.js-syntax. Deze tool is specifiek bedoeld voor educatieve doeleinden, om beginnende ontwikkelaars te ondersteunen. In tegenstelling tot FermaT ligt de focus uitsluitend op een visualisatie van de control flow.

## 2.6. High Level Assembler

High Level Assembler (HLASM) wordt gezien als een Second Generation Language (2GL), de instructies in HLASM komen rechtstreeks overeen met processorcommando's. Er zijn dus geen abstracties zoals functies of objecten. Deze abstracties zijn wel terug te vinden in talen met een hoger generatieniveau zoals COBOL. HLASM biedt daardoor veel controle tot het fysieke systeem, maar dat maakt de code vaak moeilijker te lezen en te onderhouden (Kornelis, 2003).

De structuur van HLASM-code wordt bepaald door instructies en labels. Om een subroutine aan te roepen, kun je gebruikmaken van instructies zoals BAL of BAS. Deze instructies slaan het retouradres (het adres van de instructie na de call) op in een register. Om terug te keren vanuit een subroutine kun je gebruikmaken van de instructie BR (Branch to Register), met deze instructie spring je naar het adres in het meegegeven register (Ward, 2013).

Op de IBM 370- en z/OS-mainframes is er geen hardware-stack. In plaats daarvan wordt er gebruikgemaakt van een gelinkte lijst van save areas. Een save area is een opslagruimte waarin

kopieën van alle registerwaarden en de verwijzingen naar de volgende en vorige save area zijn opgeslagen. De control flow is dus gedefinieerd in de registers en niet in een stack. Daardoor is het de taak van de ontwikkelaar om de calls en de returns, van en naar subroutines, zelf goed af te handelen door middel van de registerwaarden (Ward, 2013).

HLASM-code heeft vaste labels en instructies om de control flow van het programma te bepalen. Voor het aanroepen heb je instructies zoals BAL of BAS en voor het terugkeren heb je bijvoorbeeld BR, hierdoor kan de structuur van het programma vastgesteld worden. Dit biedt de mogelijkheid om op een geautomatiseerde manier HLASM-code statisch te analyseren en te visualiseren naar een control-flowchart in Mermaid.js-syntax.

## 2.7. Control Flow in High Level Assembler

High Level Assembler (HLASM) wordt gezien als een Second Generation Language (2GL), de instructies in HLASM komen rechtstreeks overeen met processorcommando's. Er zijn dus geen abstracties zoals functies of objecten. Deze abstracties zijn wel terug te vinden in talen met een hoger generatieniveau zoals COBOL. HLASM biedt daardoor veel controle over het fysieke systeem, maar dat maakt de code vaak moeilijker te lezen en te onderhouden (Kornelis, 2003).

Om de control-flow van HLASM te definiëren kan er gefocust worden op bepaalde elementen die sprongen of vertakkingen maken:

- Een directe sprong: Instructies zoals B (Branch) sturen het programma direct naar een ander codeblok (*High Level Assembler for z/OS z/VM z/VSE: HLASM V1R6 Programmer's Guide*, g.d.).
- Een conditionele sprong: Door middel van vergelijkingen (C voor Compare) of condities (BE voor Branch on Equal) verandert de route van het programma (*High Level Assembler for z/OS z/VM z/VSE: HLASM V1R6 Programmer's Guide*, g.d.).
- Een subroutine: Bij het aanroepen van een subroutine worden instructies zoals BAL of BAS gebruikt. Het terugkeren vanuit een subroutine gebeurt via een BR-instructie (Ward, 2013).

Deze specifieke instructies kunnen een idee geven van de control-flow van het programma.

## 2.8. JavaScript Parsing

Binnen HLASM wordt er gewerkt met een vaste kolom-gebaseerde indeling. Elk onderdeel van de code staat op een vaste plek. Elke kolom

heeft een eigen functie: kolommen 1-8 zijn voor labels, kolommen 10-14 voor de instructie en vanaf kolom 16 staan de operanden. Na kolom 71 wordt alles beschouwd als commentaar of wordt het genegeerd (*HLASM VIR6 Language Reference*, [g.d.](#)).

JavaScript is zeer nuttig om tekst te parsen, het is namelijk erg goed in het doorzoeken en herkennen van patronen (Flanagan, 2011). Voor de PoC is een regel-voor-regel aanpak voldoende. Hierbij zal de parser elke regel doorlopen op basis van de vaste kolomposities. Zo kunnen de labels en instructies herkend worden die bijdragen aan de control-flow. De verzamelde elementen worden vervolgens gemapt naar Mermaid.js-elementen.

## 2.9. Mermaid.js en flowcharts

Met Mermaid.js worden flowcharts en andere structuren vanuit tekst automatisch gevisualiseerd ([sveidqvist\\_jain\\_2021](#)). Dit maakt het makkelijk om een mapping te maken van HLASM-elementen naar een visuele representatie. Het gebruik van flowcharts om complexe software begrijpelijk te maken is een bekend middel in de informatica (Ensmenger, 2016).

Een flowchart maakt gebruik van blokken en pijlen om een route te definiëren. In de context van informatica worden flowcharts vaak gebruikt om de route van een programma weer te geven (Charntaweechun & Wangsiripitak, 2006). In het geval van HLASM kunnen bijvoorbeeld spronginstructies worden weergegeven als pijlen die het programma brengen naar een ander codeblok.

## 3. Methodologie

Deze bachelorproef is een toegepast en technisch onderzoek waarin een oplossing wordt gezocht voor het onderhouden en begrijpen van legacy HLASM-broncode. Het onderzoek start met een literatuurstudie naar het probleem-domein en het oplossingsdomein. Ook wordt de control flow van HLASM-broncode bestudeerd, waarna een subset van representatieve HLASM-instructies en labels geselecteerd worden. Na de literatuurstudie start de ontwikkeling van een technische proof-of-concept (PoC) die de geselecteerde subset visueel omzet in een control-flowchart in Mermaid.js-syntax.

### 3.1. Fase 1: Oriëntatie en Literatuurstudie

Tijdens deze fase wordt gefocust op het aanscherpen van de probleemstelling en het oplossingsdomein. Dit gebeurt door middel van een literatuurstudie waarin dieper wordt ingegaan op de deelvragen:

- Wat is het belang van de mainframecompu-

ter?

- Waarom is er nood aan het onderhouden van HLASM-code?
- Wat zijn de moeilijkheden bij het begrijpen van assemblytalen voor beginnende ontwikkelaars?
- Wat is de effectiviteit van visualisatietechnieken voor het sneller begrijpen van programma's?
- Wat zijn de bestaande visualisatietools binnen High Level Assembler?
- Welke HLASM-instructies zijn van noodzaak om de control-flow van de code in kaart te brengen?
- Wat is de beste manier om de kolom-gebaseerde structuur van HLASM te verwerken met een JavaScript-parser?
- Hoe vertaal je de vertakkingen in de HLASM-code naar correcte Mermaid.js-diagrammen?

Deze fase loopt van 20 december 2025 tot 1 maart 2026 en resulteert in een uitgewerkte literatuurstudie en methodologie. Het resultaat bevat concreet een theoretisch onderzoek naar de probleemstelling en oplossingsdomein, en een afgebakende scope van HLASM-instructies en labels voor de verdere ontwikkeling van de technische oplossing.

### 3.2. Fase 2: Requirementsanalyse

Tijdens de literatuurstudie (tussen januari 2026 en maart 2026) wordt een requirementsanalyse uitgevoerd. Het doel van deze fase is om concrete en meetbare eisen voor de proof of concept en de evaluatie vast te leggen.

Binnen deze fase beginnen we eerst met het specifiek definiëren van de doelgroep. Uit de doelgroep wordt gezocht naar 5 à 10 deelnemers die instaan voor de evaluatie van de PoC. Voor de evaluatie worden ook HLASM-codefragmenten verzameld die typische control-flowconstructies hebben zoals labels, conditionele takken en subroutines-aanroepen.

Ook worden de succescriteria van de PoC vastgesteld:

- correcte detectie van de geselecteerde HLASM-instructies en labels,
- correcte omzetting naar Mermaid.js-flowcharts,
- meetbare tijdswinst bij het begrijpen van de control flow in vergelijking met een manuele analyse.

Door deze fase zal de implementatie en evaluatie van de PoC doelgericht uitgevoerd worden.



### 3.3. Fase 3: Implementatie en Evaluatie

Tijdens de implementatie- en evaluatiefase wordt de proof-of-concept uitgewerkt en iteratief geëvalueerd. Deze fase vindt plaats tussen 2 maart 2026 en 4 mei 2026. De ontwikkeling gebeurt volgens een scrum-werkwijze om de effectiviteit van de PoC te waarborgen. Met behulp van de subset die in de vorige fase werd gedefinieerd, wordt een JavaScript-gebaseerde parser ontwikkeld die HLASM-code statisch analyseert. De parser identificeert instructies en labels aan de hand van patroonherkenning in de structuur van HLASM-broncode en zet deze via een mapping om naar Mermaid.js-syntax.

Concreet worden tijdens deze fase de volgende aspecten ontwikkeld:

- de logische opbouw van de parser op basis van patroonherkenning in de HLASM-broncodestructuur;
- een mapping van de geselecteerde HLASM-instructies en labels naar Mermaid.js-elementen.

Tijdens de ontwikkeling wordt de PoC iteratief geëvalueerd via een vergelijkende studie met traditionele, handmatige code reviews door 5 à 10 vrijwillige deelnemers. De evaluatie loopt tijdens de implementatie, van 1 april 2026 tot 4 mei 2026. De deelnemers worden gevraagd de functionaliteit van HLASM-broncode te bepalen aan de hand van zowel de technische als de traditionele methode. De resultaten worden geanalyseerd op basis van de tijd die nodig was om de functionaliteit van de HLASM-broncode te identificeren en de juistheid van de beschreven control flow. Aan de hand van deze metingen wordt bepaald of de PoC aan de gedefinieerde succescriteria voldoet.

Deze fase resulteert in een werkende PoC die HLASM-broncode omzet naar een flowchart in Mermaid.js-syntax. Daarnaast worden evaluatieresultaten verkregen die inzicht geven in de effectiviteit van de technische oplossing.

### 3.4. Fase 4: Resultaat en Verdediging

De laatste fase loopt van 4 mei 2026 tot juni 2026. In deze fase worden de onderzoeksresultaten en evaluatieresultaten verwerkt tot een conclusie.

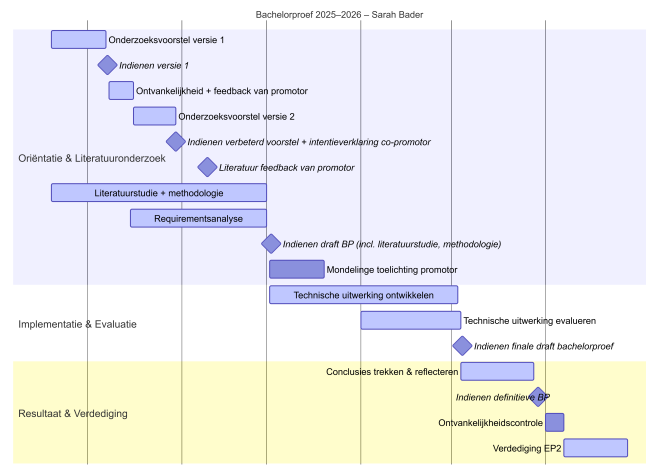
Specifiek worden de volgende aspecten besproken:

- de sterktes en beperkingen van de PoC;
- mogelijke uitbreidingen voor toekomstig onderzoek.

Deze fase resulteert in de finale bachelorproef die wordt ingediend op 29 mei 2026, de verdediging en de proof-of-concept met de bijbehorende documentatie.

### 3.5. Tijdsplanning

Elke fase heeft een tijdsperiode waarin de specifieke resultaten moeten worden opgeleverd voor de volgende fase. Zoals te zien in figuur 1, zijn ook de administratieve deliverables opgenomen.



Figuur 1: Gantt-diagram met een tijdsindicatie per onderzoeksfase.

## 4. Verwacht resultaat, conclusie

Het verwachte resultaat van deze studie is een proof of concept (PoC), die in staat is een subset van representatieve HLASM- en labels om te zetten in visuele stappen geïnterpreteerd in Mermaid.js-syntax. De PoC brengt door middel van statische HLASM-broncodeanalyse de control flow in kaart. Met deze PoC wordt verwacht beginnende mainframeontwikkelaars te ondersteunen met het begrijpen van een onbekend stuk HLASM-broncode. De evaluatie van het verwachte resultaat wordt uitgevoerd door middel van een vergelijking met de traditionele, handmatige manieren van code reviews.

## Referenties

- Baldwin, J., Myers, D., Storey, M.-A., & Coady, Y. (2009). Assembly Visualization and Analysis: An Old Dog CAN Learn New Tricks! <https://ecs.wgtn.ac.nz/foswiki/pub/Events/PLATEAU/2009Program/plateau09-baldwin.pdf>
- Charntaweekhun, K., & Wangsiripitak, S. (2006). Visual Programming using Flowchart. 2006 International Symposium on Communications and Information Technologies, 1062-1065. <https://doi.org/10.1109/ISCIT.2006.339940>

- Ensmenger, N. (2016). The Multiple Meanings of a Flowchart. *Information & Culture*, 51(3), 321–351. Verkregen januari 25, 2026, van <http://www.jstor.org/stable/44667617>
- Feigenspan, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012). Measuring programming experience. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 73–82. <https://doi.org/10.1109/ICPC.2012.6240511>
- Flanagan, D. (2011). *JavaScript: The Definitive Guide* (6th). O'Reilly Media.
- High Level Assembler for z/OS z/VM z/VSE: HLASM V1R6 Programmer's Guide* [SC26-4941-07]. (g.d.). IBM Corporation.
- HLASM V1R6 Language Reference* [SC26-4940-06]. (g.d.). IBM Corporation.
- Kawash, J. (2024). Where Do Students Struggle Most in a First Course on Assembly Language? *Proceedings of the 26th Western Canadian Conference on Computing Education*. <https://doi.org/10.1145/3660650.3660652>
- Kornelis, A. F. (2003). HLASM - Why assembler? <https://bixoft.nl/english/why.htm>
- Sagers, G., Ball, K., Hosack, B., Twitchell, D., & Wallace, D. (2018). The Mainframe Is Dead. Long Live the Mainframe! *AIS Transactions on Enterprise Systems*, 2(1). <https://www.aes-journal.com/index.php/ais-tes/article/view/6>
- Shaik, S. (2024). Importance of Mainframe in Modern Era of Technologies. *Journal of Engineering and Applied Sciences Technology*, 1–3. [https://doi.org/https://doi.org/10.47363/jeast/2024\(6\)257](https://doi.org/https://doi.org/10.47363/jeast/2024(6)257)
- Storey, M.-A. (2005). Theories, methods and tools in program comprehension. *IEEE Software*, 22(3), 36–45.
- Waites, S., & Ketterer, J. (2013). Lack of Mainframe Programmers a Critical Issue for IT Organizations A Review of the Literature. *International Journal of Business, Humanities and Technology*, 3(7). [https://ijbht.thebrpi.org/journals/Vol\\_3\\_No\\_7\\_September\\_2013/3.pdf](https://ijbht.thebrpi.org/journals/Vol_3_No_7_September_2013/3.pdf)
- Ward, M. (2013). Assembler restructuring in FermaT. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 147–156. <https://doi.org/10.1109/SCAM.2013.6648196>
- Ward, M. (2001). The FermaT assembler re-engineering workbench. *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 659–662. <https://doi.org/10.1109/ICSM.2001.972783>
- Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10), 951–976. <https://doi.org/10.1109/TSE.2017.2734091>
- Zaytsev, V. (2020). Journal of Object Technology Published by AITO -Association Internationale pour les Technologies Objets Vadim Zaytsev. Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler. *Journal of Object Technology*, 19(1), 1–22. <https://grammarware.net/text/2020/hlasm.pdf>