Gymnasiearbete (100 p)
IT-Gymnasiet Göteborg
HT 2017 — VT 2018

Supervisors:
Daniel Berg, David Lundholm

Erwall, a High Level General Purpose Programming Language

15 mars 2018

Created with LaTeX

Author:
Erik Wallström

Abstract

The primary objective of this project is to implement a fully working imperative compiled high performance general purpose programming language in C, from scratch without using any external libraries or application programming interfaces. The language will be offering many modern features with a consistent syntax, so that it can ultimately replace the C language in practice. Many commonly used tools will be used, such as git, gcc, valgrind and gdb. I will go through the general compiler pipeline and implement each component step by step. I was not able to reach all the goals that was planned in the beginning, but the project was far from a failure. The final result was a functioning, Turing complete programming language that could used for any computing task in theory. It already offers many improvements over C, and I belive that it will become a complete replacement for C in the near future. The project would probably have become far more mature if I used existing tools such as a parser and lexer generator. A significant amount of time was used to implement interfaces that was missing in the C standard library, which means that I probably would have had the time to implement more features for Erwall if I used a scripting language such as python.

Keywords: Erwall, Programming, Language, C, Compiler

Innehåll

1 Inledning

Dagens informationssamhälle är helt beroende av datorer. Fler och fler jobb automatiseras, och behovet att kunna skapa och utveckla felsäkra program som ska kunna köras på dem är större än någonsin. För att skapa ett datorprogram måste man på något sätt kunna kommunicera med datorer för att framföra vad man vill göra. Datorer förstår dock endast maskinkod; ettor och nollor. Man insåg redan på 1950-talet att det var ohållbart för människor att skriva maskinkod för hand, eftersom att det var oerhört tidskrävande och ledde ofta till fel. Därför skapade man så kallade programmeringsspråk; kod som består av text och siffror som är betydligt lättare för människor att hantera. Under åren har det skapats väldigt många olika programmeringsspråk.

Man kan dela upp programmeringsspråk i flera kategorier, bland annat dynamiska och statiska, generella och specifika, imperativa och deklarativa med mera. Ett statiskt programmeringsspråk översätts till maskinkod med hjälp av ett program som kallas för en kompilator. Dynamiska språk använder sig istället av interpretatorer, som är som virtuella maskiner som kör koden. Generella programmeringsspråk är språk som kan användas till vad som helst, till exempel datorspel, textredigeringsprogram eller webbläsare. Specifika programmeringsspråk är endast avsedda för en enda uppgift, till exempel statistik eller specialeffekter. Om du använder ett imperativt programmeringsspråk måste du steg för steg förklara för datorn hur den ska göra för att lösa en uppgift, medan du endast behöver säga vad du vill få löst i ett deklarativt språk. Man pratar också ofta om hög- och lågnivåspråk. Ett lågnivåspråk är väldigt lik maskinkod, och kan endast göra det som processorn kan göra, medan högnivåspråk har många abstraktioner som gör att man kan ha variabler, funktioner och procedurer som en processor inte direkt kan förstå.

1.1 Bakgrund

Programmering har varit min hobby i nästan 5 år, och under den tiden har jag testat ett stort antal av de existerande programmeringsspråken. Dock så har jag ännu inte hittat något språk som jag anser vara perfekt; antingen saknas funktioner, eller så finns det onödiga och överflödiga funktioner som jag ogillar, med mera. Därför har jag valt att skapa ett nytt imperativt statiskt högnivåprogrammeringsspråk som kan användas i praktiken. Genom att skapa ett eget programmeringsspråk så kan jag anpassa och förbättra vissa delar så att det blir det perfekta programmeringsspråket för mig. Projektet kommer att ha öppen källkod samt en fri upphovsrättslicens.

Jag har gått kursen Programmering 1, och går just nu i Programmering 2. Tidigare har jag skapat en interpretator för programmeringsspråket *Brainfuck*, en textredigerare och utvecklingsmiljö med syntax highligting för programmeringsspråket *Lua*. Jag har även erfarenhet av många programmeringsspråk, inklusive men inte begränsat till *Java*, *C*, *Lua*, *C++* och *Ruby*. Allt detta gör att jag bedömer att jag är tillräckligt kompetent för att klara detta projekt, även om det är väldigt ambitiöst för en person.

Mycket av detta nya språk kommer att bygga på programmeringsspråket C, som är mitt favoritspråk och det språk som jag har mest erfarenhet av. C skapades på 1970-talet och är ett statiskt högpresterande imperativt högnivåspråk, och är idag ett av det mest använda programmeringsspråken. Många andra programmeringsspråk är skapade i C, och det finns mycket resurser och information om det eftersom att det är så pass populärt. Detta gör C till ett väldigt bra språkval för just detta projektet.

1.2 Syfte

Syftet med detta projekt är att få fram ett fungerande programmeringsspråk som på sikt kan ersätta programmeringsspråket C. Språket ska ha en del moderna egenskaper, göra det enklare för programmerare att skapa högpresterande och felsäkra program, samt göra det roligare att programmera.

Detta projekt ska dessutom visa att det inte är så märkvärdigt att skapa ett programmeringsspråk; många programmerare klagar på programmeringsspråk, men ytterst få försöker förbättra dem. Programmeringsspråk brukar ha en speciell status, speciellt bland nybörjare, på grund av mentaliteten att man inte ska återuppfinna det som redan finns, att det är för svårt, samt att man inte ska byta ut det som redan fungerar. Många tycker därför att det är orimligt att skapa ett nytt programmeringsspråk, och tänker inte på att det bara rör sig om en viss standard som är satt, eller att en kompilator är ett program som helt enkelt tolkar och översätter koden du skriver till maskinkod.

Detta projekt kommer även fungera som en övning för att träna på avancerad programmering, strukturering och arbete av stort projekt och göra en till en bättre programmerare. För att vara en bra programmerare måste man förstå hur programmeringsspråk är uppbyggda och fungerar, vilket gör att detta projekt är perfekt för ändamålet.

1.3 Frågeställningar

- Vilka delar är en kompilator uppbyggd av och varför?
- Hur bra fungerar C som ett programmeringsspråk för att skapa en kompilator?
- Är C ett bra mål att generera kod till?
- Hur ska man planera ett projekt för att tillåta enkel felsökning och tillägg av nya funktioner?

1.4 Kravspecifikation

Generella krav:

- Logisk och konsekvent syntax

  Många språk har ologisk och inkonsekvent form, där till exempel funktionsdeklarationer inte ser likadana ut beroende på vart de är deklarerade, och vissa variabeldeklarationer ser ut som matematiska uttryck. Funktionsargument ser generellt sett inte ut som variabler, fastän de i praktiken är exakt likadana. Detta gör det jobbigt för programmerare, och därför ska sådana specialfall inte existera i Erwall.

- Explicita deklarationer

  När du låter kompilatorn gissa vad du vill att den ska göra kallas det för implicita deklarationer. Då skriver du inte ut exakt vad du vill att den ska göra. Detta gör att det kan bli svårare att förstå koden, och kan leda till logiska fel. Därför ska Erwall endast ha explicita deklarationer.

- Högpresterande

Vissa programmeringsspråk prioriterar hastigheten som man kan skapa program med framför hur snabbt programmet körs. Man kan tänka att man föredrar kvantitet före kvalitet. Erwall kommer dock att prioritera hastigheten som programmet körs med, så att kvalitén på produkten programmeraren skapar blir så bra som den kan bli.

Några specifika specifikationer på språket kommer vara följande:

- Stöd för *nested* kommentarer och funktioner

  I många programmeringsspråk kan du inte ha kommentarer i kommentarer, eller funktioner i funktioner. Detta är ologisk, och kan i många fall göra det jobbigare för programmerare.

- Garanterade *tail call* optimeringar

  Vissa språk så som C har inte garanterade *tail call* optimeringer, som ofta krävs för att programmera på ett funktionellt sätt; programmet kommer att krasha efter ett tag om man inte har det. Erwall kommer att stödja detta.

- Kompatibelt med C

  Eftersom att C är ett så populärt och moget språk finns det väldigt många bibliotek och gränssnitt skrivna i C. Genom att göra Erwall kompatibelt med C kan man använda dessa redan skapade bibliotek istället för att skapa allt från grunden, vilket är orimligt.

- Strikt typsystem med riktiga konstanter

  Många språk har inte strikta typsystem. Detta gör att till exempel heltal implicit kan omvandlas till decimaltal. Detta kommer inte vara tillåtet i Erwall.

- Inget odefinerat beteende

  I till exempel C kan man skriva kod som har odefinerat beteende; ingen vet vad som kommer hända. Detta gör att det blir lätt att skriva felaktig kod. I Erwall kommer all kod som kompileras vara korrekt och definerad, för att underlätta för debugging som ofta tar väldigt lång tid.

- Listor som första klassens medborgare

  En entitet som stödjer vanliga operationer, så som att de kan fungera som argument, returneras och tilldelas till variabler, kallas för första klassens medborgare. I många språk är listor inte första klassens medborgare, vilket gör att de sällan används. Istället används bibliotek och andra gränssnitt. Erwall kommer att ha stöd för dem, så att man slipper detta. Listor behövs för nästan alla projekt, och bör vara inbyggda i språket.

- Tagged unions

  En union är en datatyp som kan ha en av flera definerade typer. I C vet man inte vilken av typerna unionen har. Detta gör att man själv måste hålla koll på det. Erwall kommer att ha detta inbyggt, så att man själv slipper göra det.

1.5 Metod och material

Fokus lades på att skapa en fungerande kompilator. Språket kommer att anpassas och utvecklas efter hand. Detta gjordes då ett av målen var att språket ska kunna användas i praktiken. Hade det varit ett mer teoretiskt arbete skulle fokus istället läggas på språk specifikationer och en standard. Projektet bygger på research utifrån tekniska artiklar, föreläsningar samt wikipedia. Utifrån informationen som samlades ihop skapades en projektplan som sedan följdes. Genom detta kunde frågeställningen "vilka delar är en kompilator uppbyggd av' besvaras.

En *lexer* skapades först. Detta är den första komponenten i de flesta kompilatorer. Denna komponent delar upp koden i så kallade *tokens*, som används för att särskilja speciella kodord med identifierare, som är namn på variabler och funktioner som programmeraren kan definera. Därefter implementerades en *parser*, som konstruerar ett abstrakt syntaxträd (AST) av *tokens*, så att man tydligt kan se hierarki och gruppereringar. Efter det skapades en semantisk analysator, för att kontrollera om koden programmeraren har skrivet är logisk, till exempel om man verkligen kan addera text med nummer. Den semantiska analysator konstruerar även en symboltabell. Sedan skapades en kodgenerator, som tar trädet och symboltabellen och genererar C kod. Till sist skapades ett typsystem. Vid implementationen av kodgeneratorn kunde svaret till frågan om C är ett bra mål att generera kod till besvaras.

Alla komponenter skrevs på en dator med Arch Linux i programmeringsspråket C (gnu-c11) med hjälp av textredigeringsprogrammet *vim* och kompilatorn *gcc*, för att kunna besvara frågan om C fungerar bra för att skapa en kompilator. För felsökning och kontroller användes verktygen *gdb* och *valgrind*, som är standard för debugging av C program i Linux. Inga externa bibliotek eller *application programming interfaces* (API) har använts; allt är skrivet från grunden. Detta gjordes så att man verkligen förstår processen. Om färdigskrivna gränssnitt hade använts hade man inte kunnat anpassa det lika bra, och mycket av implementationen hade då varit färdiggjort och på så sätt hade det varit svårt att besvara många av frågeställningarna.

Under projektets gång användes det distribuerade versionhanteringssystemet *git* för att hålla ordning på utveckling, ändring av kod och för att förenkla felsökning. Dessutom möjliggjorde detta för andra människor att bidra med förbättringar och ändringar, då Erwalls kompilator har öppen källkod.

2 Resultatredovisning

2.1 Arbetsgången

Under de första veckorna lades majoriteten av tiden ner på planering, efterforskning om ämnet, idéer samt design av syntax. Ett github-projekt skapades, med en informativ sida om språket.

2017–10–17

Första prototypen av en *lexer* samt alla grundläggande strukturer blev klara idag. *Lexern* har enkel felrapportering där både rad och kolumn där felet hittades visas, operatorer, nyckelord, identifierare samt sträng-, *boolean*- och nummerlitteraler, flerradiga kommentarer som kan vara nested. Eftersom att standard-biblioteket i C saknar mycket behövdes

funktioner som bland annat dynamiska generiska listor, hantering av färgkoder för Linux-terminalen samt en abstraktion av filhanteringgränssnittet implementeras.

2017–10–19

Idag blev första *parser*-prototypen färdig. Ett gränssnitt för AST:s blev skrivet, och logisk utmatning lades till så att man enkelt och grafiskt kan se hur trädet är uppbyggt. Parsern är en så kallad *recursive descent parser*, som använder tokens från *lexern* samt AST-gränsittet för att konstruera ett träd. Just nu stödjer det bland annat hantering av funktioner och block, med logiska felrapporter. Ett problem med syntaxen för flerradiga kommentarer som orsakade fel resultat i vissa specifika fall hittades. Därför ändrades koden i *lexern* för att lösa detta.

2017–10–20

---

*Parser*n stödjer nu numeriska uttryck med alla vanliga matematiska operatorerna, funktionsanrop samt variabeldeklarationer. *Lexern* stödjer nu även *bitwise*-operatorer.

2017–10–22

Nu stödjer *parsern* även *if*-satser och booleanska uttryck.

2017–10–23

Idag började arbetet på semantisk analys av AST:t. Kompilatorn rapporterar nu om odefinerade variabler, funktioner och typer, samt när du anropar en funktion med inkorrekt antal argument med mera. Ett gränssnitt för *scopes* blev även skapat. Dessutom gjordes det så att *parsern* nu kan hantera typdeklarationer och *return*-satser. Idéer om syntax för listor samt referenser för språket blev påkommna.

2017–10–28

Tester för logiska, numeriska operationer och funktionsdeklarationer har nu framgångsrikt blivit implementerade. *Parser*n stödjer nu även typomvandlingar.

2017–10–29

Idag lades kod till så att *parsern* klarar av *elseif*- och *else*-satser, samt externa funktionsanrop. Semantiska kontroller för *main*-funktionen blev även tillagda.

2017–11–01

Experimentell C-kodgenerering har nu lagts till. En abstraktion av C-stränggränssnittet blev skapat eftersom att standardgränssnittet inte var tillräckligt användbart för projektets ändamål.

2017–11–02

Kompilatorn stödjer nu argument för att stödja frivillig utskrivning av AST och tokens med mera. C har inget sådant bibliotek i standarden, vilket ledde till att en API för *POSIX*-baserade kommandoradsargument blev skriven från grunden.

2017–11–03

Kompilatorn stödjer nu automatisk kompilering av den genererade C koden med hjälp av C kompilatorn *gcc*.

2017–11–04

Parsern och kodgeneratorn stödjer nu tomma typer. Generatorn skapar nu korrekt indenterad C kod, och genererar *if*-satser korrekt. Semantiska tester för externa funktionsanrop samt *memory leaks* i semantiska tester är nu fixade.

2017–11–05

Små förbättringar av funktionsgenereringen lades till.

2017–11–07

Semantisk information lades in i AST noderna, och stöd för matematiska uttryck lades till i *parsern*. Experimentell generering av lokala funktioner och funktionsanrop lades även till.

2017–11–08

En bug som gjorde att lokala variabler inte genererades korrekt blev nu fixat.

2017–11–26

Efter mer än två veckors arbete har nu hela projektet blivit omstrukturerat. Anledningen var att felrapporteringen inte var tillräckligt bra, samt att mycket blev rörigt. Det var bättre att strukurera om helt än att ändra på allt som redan var skrivet bedömdes det. Hela projektet är nu betydligt bättre strukturerat vilket gör att framtida ändringar blir enklare. Felrapporteringen är nu tydlig och färgkodad. Semantiska analyser för oanvända och oinitierade variabler fungerar nu korrekt, och *parsern* stödjer nu tomma *return*-satser.

2017–12–01

Semantisk analys för att se om funktioner returnerar korrekt fungerar nu. Några *memory leaks* som upptäcktes har nu blivit åtgärdade. Enkel optimering påbörjades, men det visade sig vara betydligt svårare än beräknat.

2017–12–21

Under veckan har implementationen av en enkelt interpretator för *compile-time execution* av kod försökts. Efter ett tag bedömdes det dock att detta var ett för stort ämne att börja på nu.

2017–12–29

Små förbättringar till kodgeneratorn och analysatorn har lagts till. Bland annat klarar kodgeneratorn nu externa funktionsanrop, *if*-satser samt typkonversioner.

2018–01–01

Kodgeneratorn har återigen blivit fixad då den fortfarande hade vissa problem. Lokala funktioner samt exponenter i uttryck fungerar nu som de ska göra.

2018–01–05

---

*Parser*n och generatorn hanterar nu även *defer*-satser.

2018–01–06

While-satser *parse*:as och generaras nu korrekt. Förbättringar till felmeddelanden lades till.

2018–01–20

Prestandatester har nu lagts till, så att man enkelt kan se hur lång tid varje komponent i kompilatorn tar. Några små fel som hittades blev även lösta.

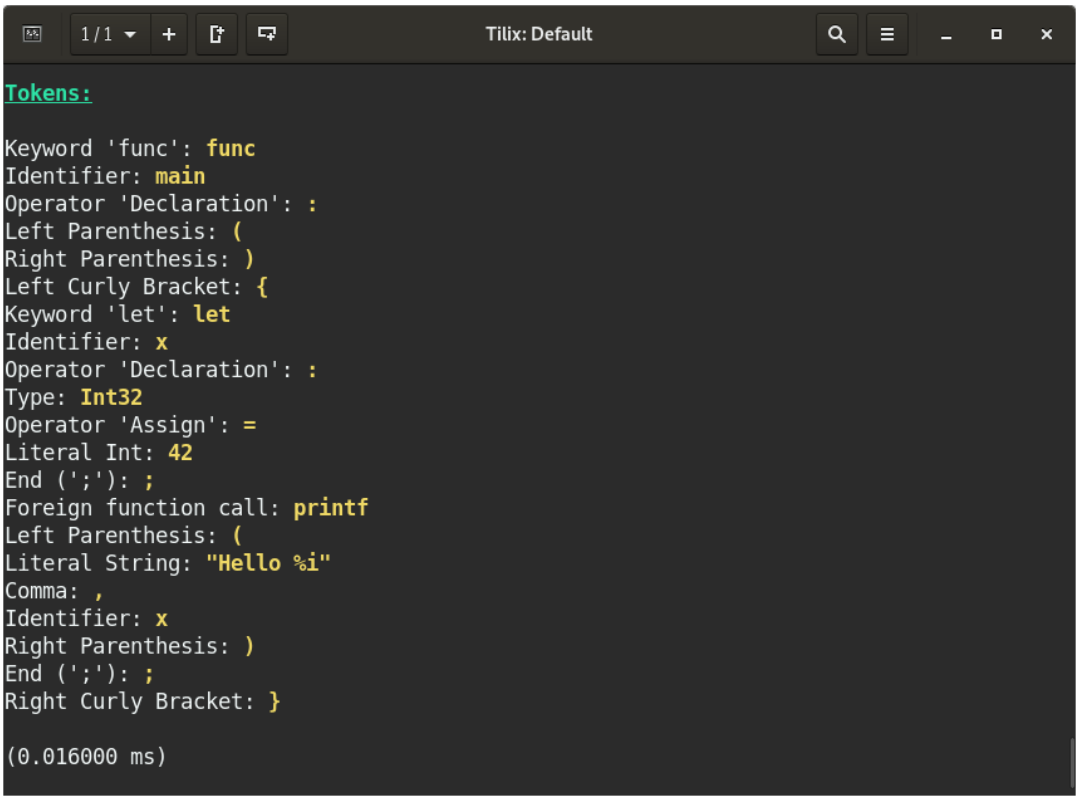
2018–01–21

Ett riktigt typsystem har nu blivit implementerat. Det kan hantera listor, pekare, funktioner, unioner, enumerationer, strukturer samt typdefinitioner.

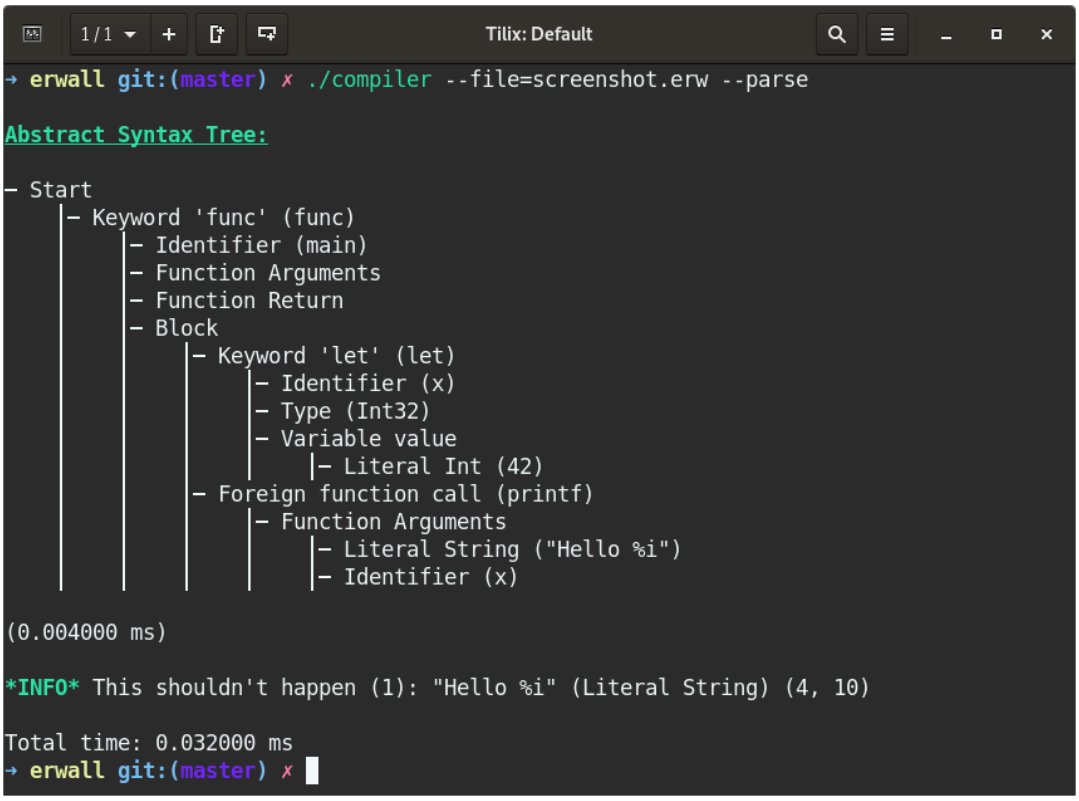
2.2 Resultatet

Slutresultatet kan hittas här: `https://github.com/ErikWallstrom/Erwall`

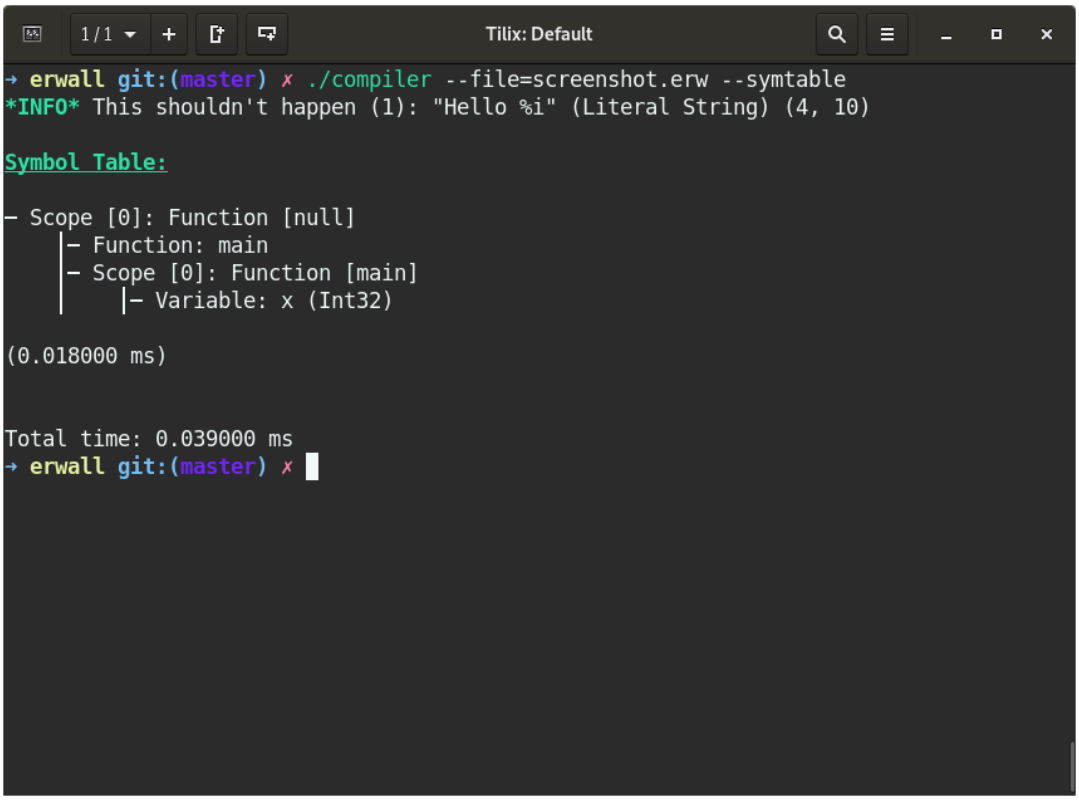
Kompilatorn och språket är långt ifrån klart. Många av specifikationerna som sattes i början av projektet han inte bli färdiga. Dock så är Erwall nu ett fungerande programmeringsspråk som jag bland annat använde en matematiklektion, vilket tyder på att det redan är användbart och på vissa sätt till och med bättre än C. Bland annat är typsystemet betydligt mer avancerat, och konstanter samt sematiska kontroller och felmeddelande är betydligt bättre i jämförelse. I teorin skulle man kunna skriva vilka program som helst i det.



Figur 1: Utdata från *lexern*



Figur 2: Utdata från *parsern*



Figur 3: Utdata från symboltabellen

Figur 4: Exempel på Erwall-kod

3 Diskussion och slutsatser

4 Källförteckning

https://medium.freecodecamp.org/the-programming-language-pipeline-91d3f449c919

https://en.wikipedia.org/wiki/Lexical_analysis

https://en.wikipedia.org/wiki/Recursive_descent_parser

https://en.wikipedia.org/wiki/Abstract_syntax_tree

https://en.wikipedia.org/wiki/Symbol_table

https://en.wikipedia.org/wiki/Semantics_(computer_science)

https://github.com/odin-lang/Odin

https://github.com/BSVino/JaiPrimer/blob/master/JaiPrimer.md

https://inductive.no/jai/

https://stackoverflow.com/questions/21945891/how-do-i-check-whether-all-code\
-paths-return-a-value

https://stackoverflow.com/questions/1825298/your-favourite-abstract-syntax\
-tree-optimization