Programski prevodioci

13 Razno

Fakultet tehničkih nauka, Novi Sad

20-21/Z

Dunja Vrbaški

Prethodno predavanje: stek mašine, java bytecode
U nastavku: resursi, predispitne

Predispitne i ispit

Ispit

Otisak

- verovatno će biti negativnih
- možda će biti pitanja sa odgovorom u slobodnoj formi
- pitanja mogu nositi različit broj bodova
- planira se trajanje od oko 30’

- pitanja sa testa za predispitne
- dodatno

Poziv na usmeni

- Bazni blokovi - odrediti broj baznih blokova, odrediti gde počinje treći bazni blok,...
- Međukod - Infiksna <-> postfiksna notacija, TAC,...
- Alokacija registara, upravljanje memorijom, aktivacioni blokovi,...
- Optimizacije – vrste, koja je primenjena u datom kodu, koja se može primeniti u datom kodu,...
- izraz <-> stek masina, java <-> bytecode
- Opšte - navedi...opiši ukratko...

compilers in the wild

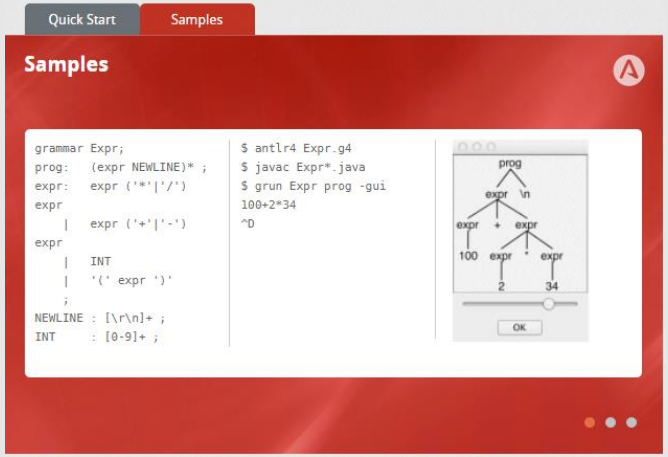
Alati za parsiranje

Razvoj novog jezika – potencijalno ubrzanje

- mnogi: yacc/bison -> ručno pisani rekurzivni spust (gcc) – brže, efikasnije, bolje informacije o greškama, lakše modifikacije,...
- ideja: na početku koristiti alat, kad je jezik stabilan ili se nastavlja sa radom na njemu razmotriti implementaciju bez alata

Bison C++, Java
ANTLR (https://www.antlr.org/)

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



LLVM

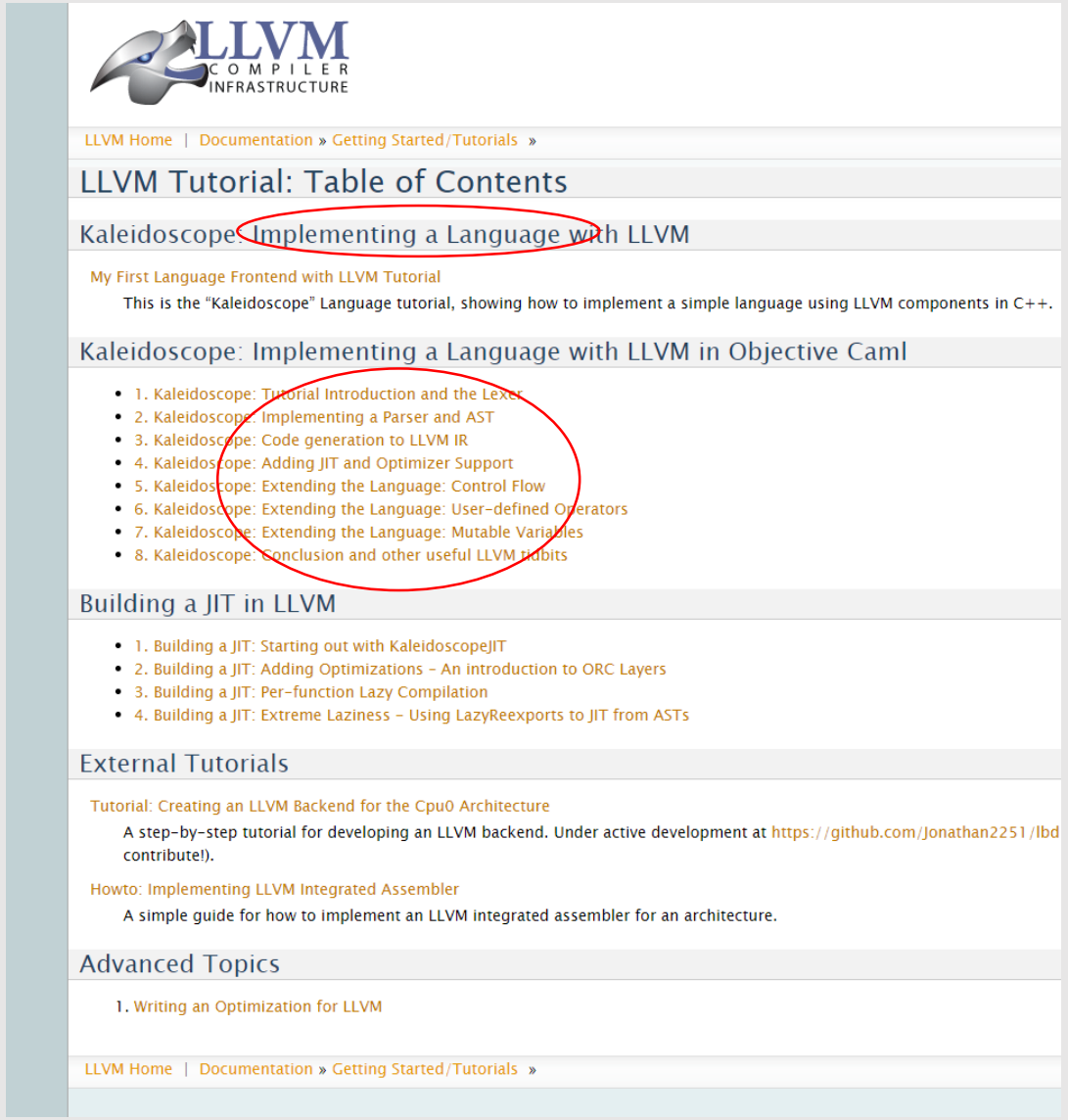
(davno bilo “Low Level Virtual Machine”)

https://llvm.org/

“compiler infrastructure”

- framework za razvoj kompajlera/programskih jezika
- open source
- različiti projekti
  - Clang – C/C++ kompajler
  - implementacija C++ SL (Standard Library)
  - optimizacije, različiti zadnji moduli – nezavisno od polaznog jezika
  - LLVM IR (LLVM bitcode / assembly lang / in-memory)
    - centar projekta
    - Static Single Assignment form (SSA)
    - beskonačno temp promenljivih (registara)
  - -...

- Swift, Rust... – LLVM back-ends; Facebook C++ Clang;…



LLVM COMPILER INFRASTRUCTURE

LLVM Home | Documentation » Getting Started/Tutorials »

LLVM Tutorial: Table of Contents

Kaleidoscope: Implementing a Language with LLVM

My First Language Frontend with LLVM Tutorial

This is the "Kaleidoscope" Language tutorial, showing how to implement a simple language using LLVM components in C++.

Kaleidoscope: Implementing a Language with LLVM in Objective Caml

1. Kaleidoscope: Tutorial Introduction and the Lexer
2. Kaleidoscope: Implementing a Parser and AST
3. Kaleidoscope: Code generation to LLVM IR
4. Kaleidoscope: Adding JIT and Optimizer Support
5. Kaleidoscope: Extending the Language: Control Flow
6. Kaleidoscope: Extending the Language: User-defined Operators
7. Kaleidoscope: Extending the Language: Mutable Variables
8. Kaleidoscope: Conclusion and other useful LLVM tidbits

Building a JIT in LLVM

1. Building a JIT: Starting out with KaleidoscopeJIT
2. Building a JIT: Adding Optimizations - An introduction to ORC Layers
3. Building a JIT: Per-function Lazy Compilation
4. Building a JIT: Extreme Laziness - Using LazyReexports to JIT from ASTs

External Tutorials

Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

A step-by-step tutorial for developing an LLVM backend. Under active development at https://github.com/Jonathan2251/lbd contribute!).

Howto: Implementing LLVM Integrated Assembler

A simple guide for how to implement an LLVM integrated assembler for an architecture.

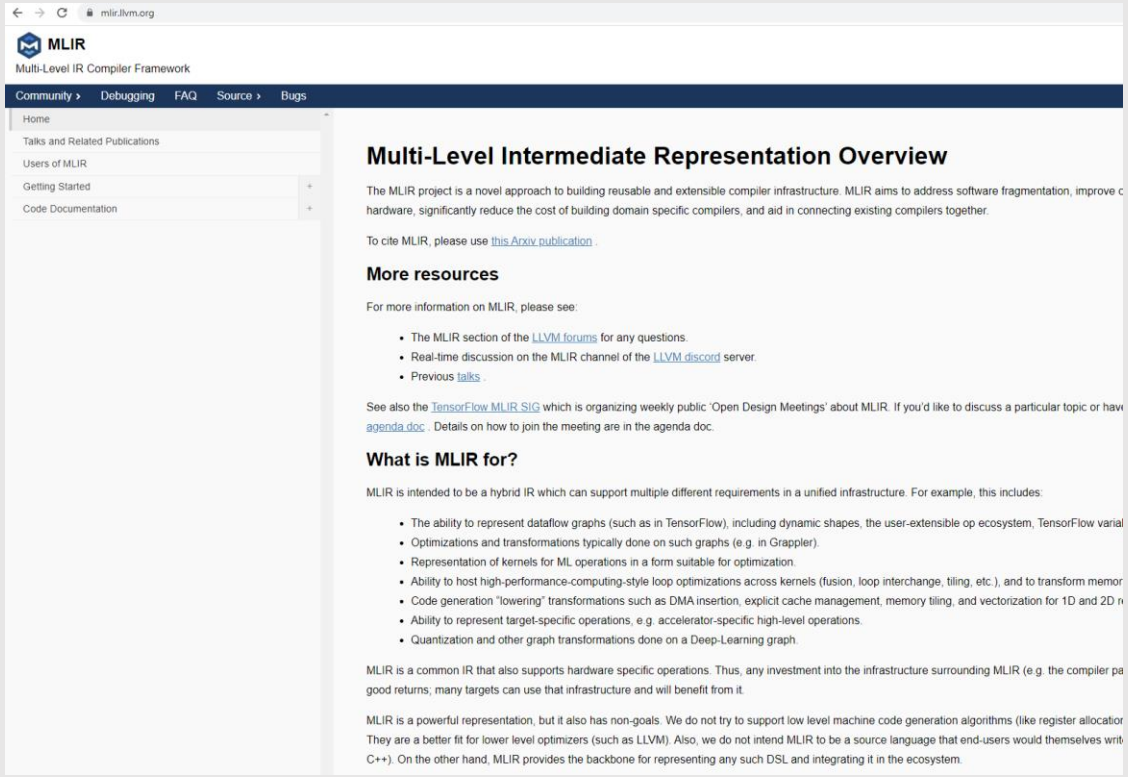
Advanced Topics

1. Writing an Optimization for LLVM

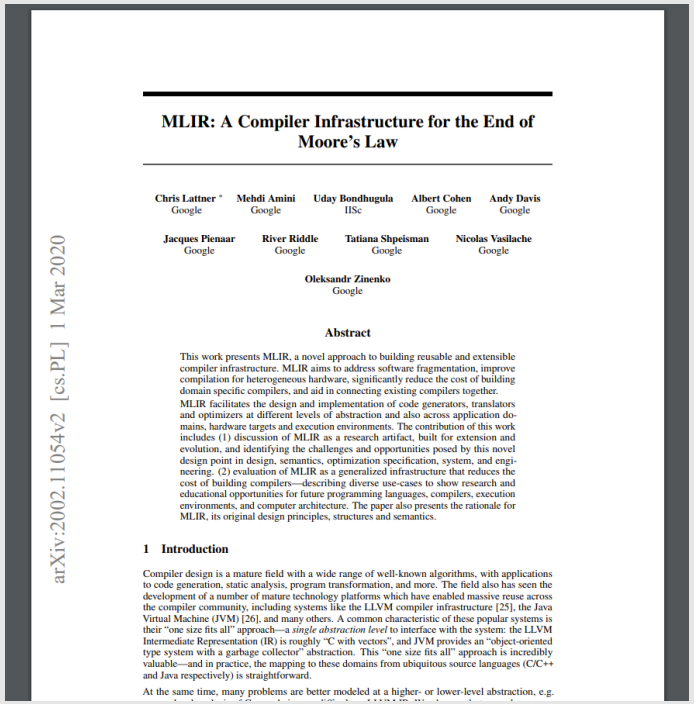
LLVM Home | Documentation » Getting Started/Tutorials »

https://llvm.org/docs/tutorial/

https://mlir.llvm.org/
https://github.com/llvm/llvm-project/tree/master/mlir/



hybrid IR
- represent dataflow graphs (such as in TensorFlow)
- optimizations typically done on such graphs (e.g. in Grappler).
- representation of kernels for ML operations
- host high-performance-computing-style loop optimizations
- graph transformations done on a Deep-Learning graph
- ...

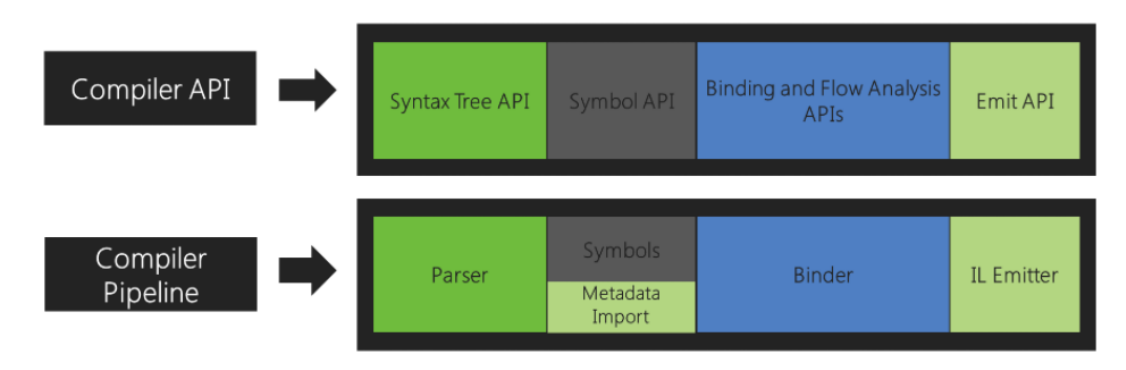


CLR

Common Language Runtime

https://docs.microsoft.com/en-us/dotnet/core/introduction

- .NET run-time okruženje
- Bogat API - .NET Compiler Platform SDK (Roslyn – kompajleri i alati za C# i VB)



Primer: `SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);`

https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis

WebAssembly

```
int sum(int a, int b) {
   return a + b;
}
```

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "sum" (func $sum))
  (func $sum (; 0 ;) (param $0 i32) (param $1 i32)(result i32)
    (i32.add
       (get_local $1)
       (get_local $0)
    )
   )
)
```

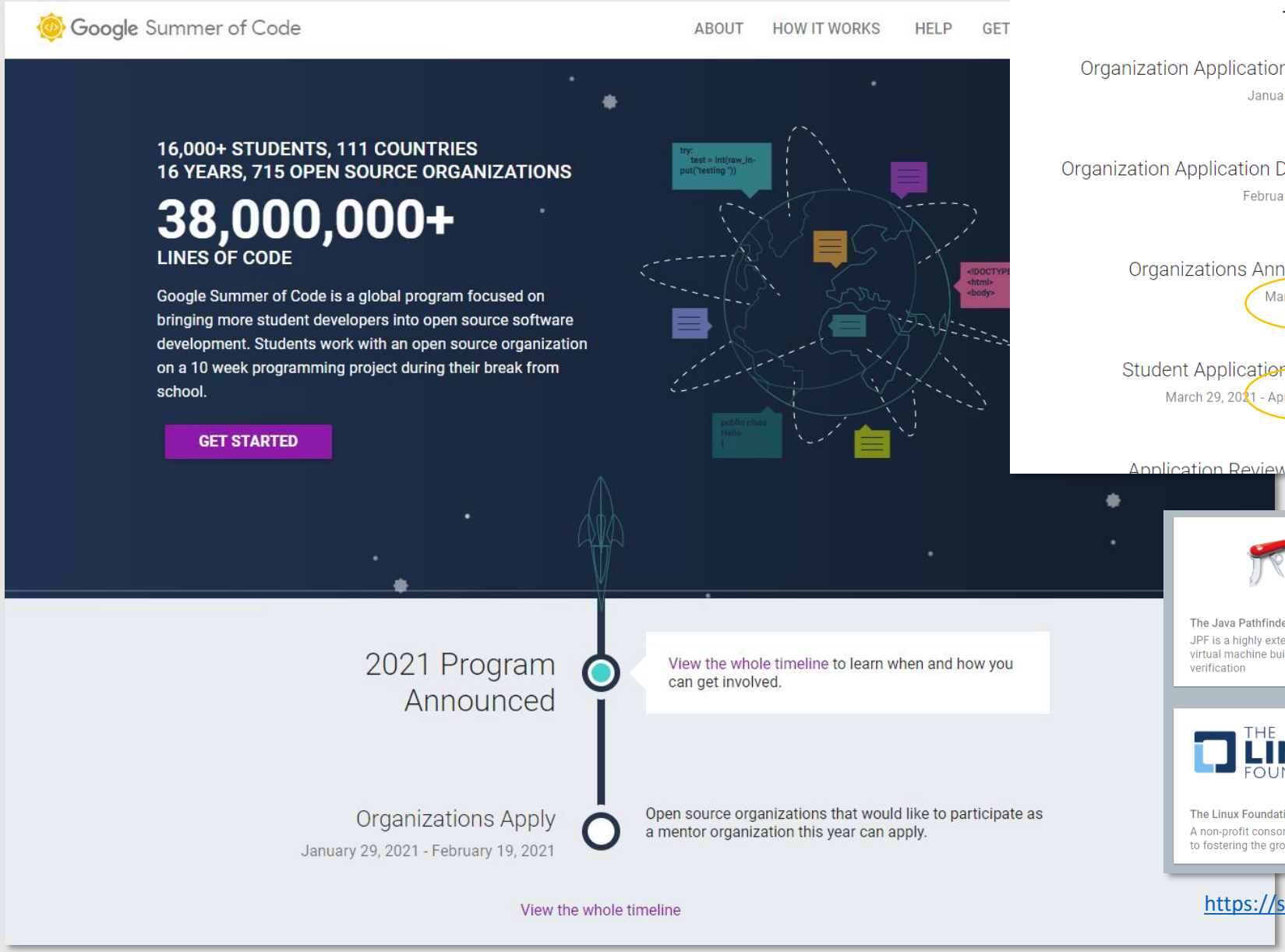
s-izrazi

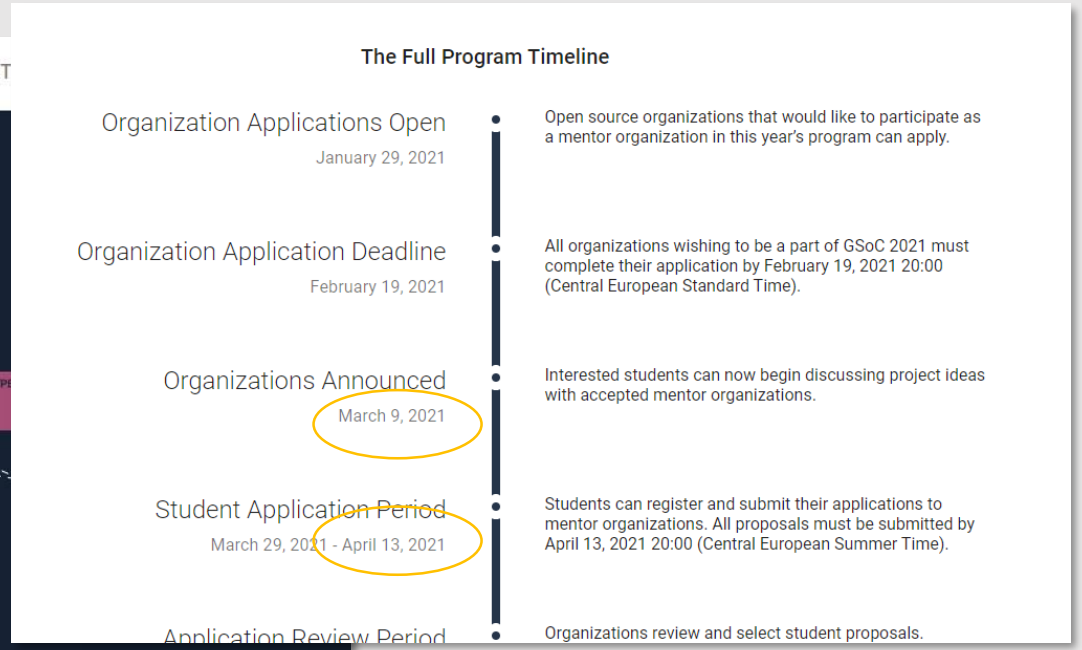
- low-level IL jezik za stek VM
- wasm (binary), wat (human readable)
- namenjen za web, prvenstveno
- svi moderni browser-i imaju podšku
- znatno brže izvršavanje nego JS – ali radi zajedno sa JS
- omogućava i da se jezici koji se tradiconalno koriste za desktop aplikacije, servise, biblioteke (C, C++, Rust...) koriste za razvoj aplikacija za web -> biblioteke i funkcije napisane u ovim jezicima koriste se na web-u

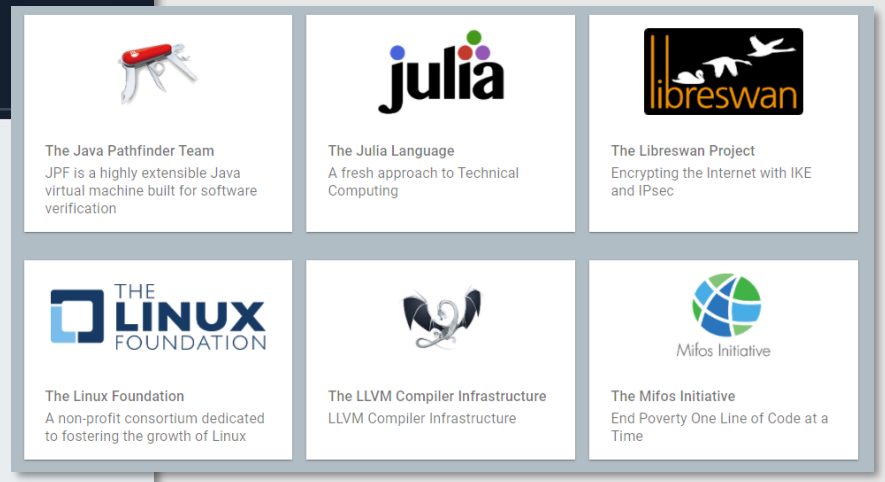
Npr:
https://emscripten.org/index.html (LLVM)
c/c++ -> wasm / js / html

https://v8.dev/ - Google JS and WebAssembly engine (primer implementacije jezika JS (ECMA) i WA)

https://summerofcode.withgoogle.com/







https://summerofcode.withgoogle.com/archive/2020/organizations/

https://www.sigplan.org/

SIGPLAN

Special Interest Group on Programming Languages

The ACM Special Interest Group on Programming Languages (SIGPLAN) explores programming language concepts and tools, focusing on design, implementation, practice, and theory. Its members are programming language developers, educators, implementers, researchers, theoreticians, and users.

Key Links

- SIGPLAN Blog
- Conferences
- Calendar
- OpenTOC
- Awards
- Research Highlights
- SIGPLAN By-laws
- SIGPLAN Cares
- SIGPLAN Officers
- Membership
- Mentoring
- Student Membership
- Travel Support
- Summer Schools
- Resources

Ad Hoc Committees

- Climate Change
- Empirical Evaluation

Conference Information

- Author Information
- Steering Committee Guidelines

SIGPLAN Chair's Inclusivity Statement

SIGPLAN Blog

Awards and Recent Recipients

Programming Languages Achievement Award (presented in 2020)

- Hans-J. Boehm

Distinguished Service Award (presented in 2019)

- Jan Vitek

Programming Languages Software Award (presented in 2020)

- Pin

Robin Milner Young Researcher Awa

- Martin Vechev

John C. Reynolds Doctoral Dissertat

- Filip Niksic, Max Planck Institu
  Combinatorial Constructions f

Most Influential POPL Paper Award (

- Saurabh Srivastava, Sumit Gu
  (for 2010): *From program verif*

Most Influential PLDI Paper Award (

- Woongki Baek (Ulsan Nationa
  (Amazon)
  (for 2010) *Green: A Framewor*
  Approximation

Most Influential ICFP Paper Award (

- David Van Horn and Matthew
  (for 2010) *Abstracting Abstrac*

Most Influential OOPSLA Paper Awa

- Lennart C.L. Kats, Eelco Visser
  (for 2010) *The spoofax language workbench: rules for declarative specification of languages and*

SIGPLAN Sponsored Conferences

- Architectural Support for Programming Languages and Operating Systems (A
- International Symposium on Memory Management (ISMM)
- Virtual Execution Environments (VEE)
- Programming Language Design and Implementation (PLDI)
- Principles of Programming Languages (POPL)
- Object-oriented Programming, Systems, Languages, and Applications (OOPS
- International Conference on Functional Programming (ICFP)
- Languages, Compilers, and Tools for Embedded Systems (LCTES)
- Principles and Practice of Parallel Programming (PPoPP)
- Systems, Programming, Languages, and Applications: Software for Humanity
- Onward!
- Dynamic Languages Symposium (DLS)
- Generative Programming: Concepts and Experiences (GPCE)
- Code Generation and Optimization (CGO)
- Software Language Engineering (SLE)
- Programming Languages Mentoring Workshop

CGO 2021 (series) /

Student Research Competition

About | Call for Papers | Submission Information

Call for Papers

The ACM Student Research Competition (SRC) offers a unique forum for undergraduate and graduate students to present their original research before
graduate students pursuing an acad
members of the ACM.

To participate in the competition, a s
a selection committee and selected
presentations (10 minutes + 5 minut
abstract and the presentation, the wi
award. In addition, the winner will be
on the ACM SRC is available at src.

Submissions in the form of an exten

- Code Generation, Translation
  security, or reliability concerns
- Efficient execution of dynamic
  emerging programming mode
- Dynamic/static, profile-guided
- Static, Dynamic, and Hybrid A
  reliability, or functional debugg
- Program characterization met
- Efficient profiling and instrume
- Novel and efficient tools
- Compiler design, practice, an
- Compiler abstraction and inte
- Vertical integration of languag
- Solutions that involve cross-la
- Deployed dynamic/static com
  platforms
- Parallelism, heterogeneity, an
- Optimizations for heterogenec

SPLASH 2020 (series) /

Student Research Competition

About | Program

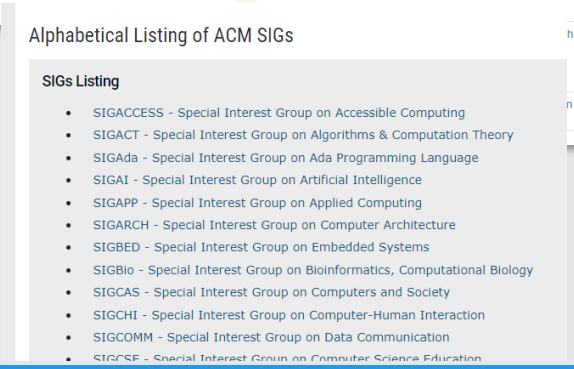
The ACM Student Research Competition (SRC), sponsored by Microsoft Research, offers a unique opportunity for undergraduate and graduate students to present their research to a panel of judges and conference attendees at SPLASH. The SRC provides visibility and exposes up-and-coming researchers to computer science research and the research community. This competition also gives students an opportunity to discuss their research with experts in their field, get feedback, and sharpen their communication and networking skills.

To participate in the competition, a student must submit a 2-page description of his or her original research project. The submitted project descriptions are peer-reviewed. Each student whose description is selected by a panel of reviewers is invited to attend the SRC competition at SPLASH and present their work.

Winners of the SPLASH competition are invited to participate in the ACM Student Research Competition Grand Finals. Submit your work and take part of the ACM Student Research Competition at SPLASH 2020!

Accepted Papers

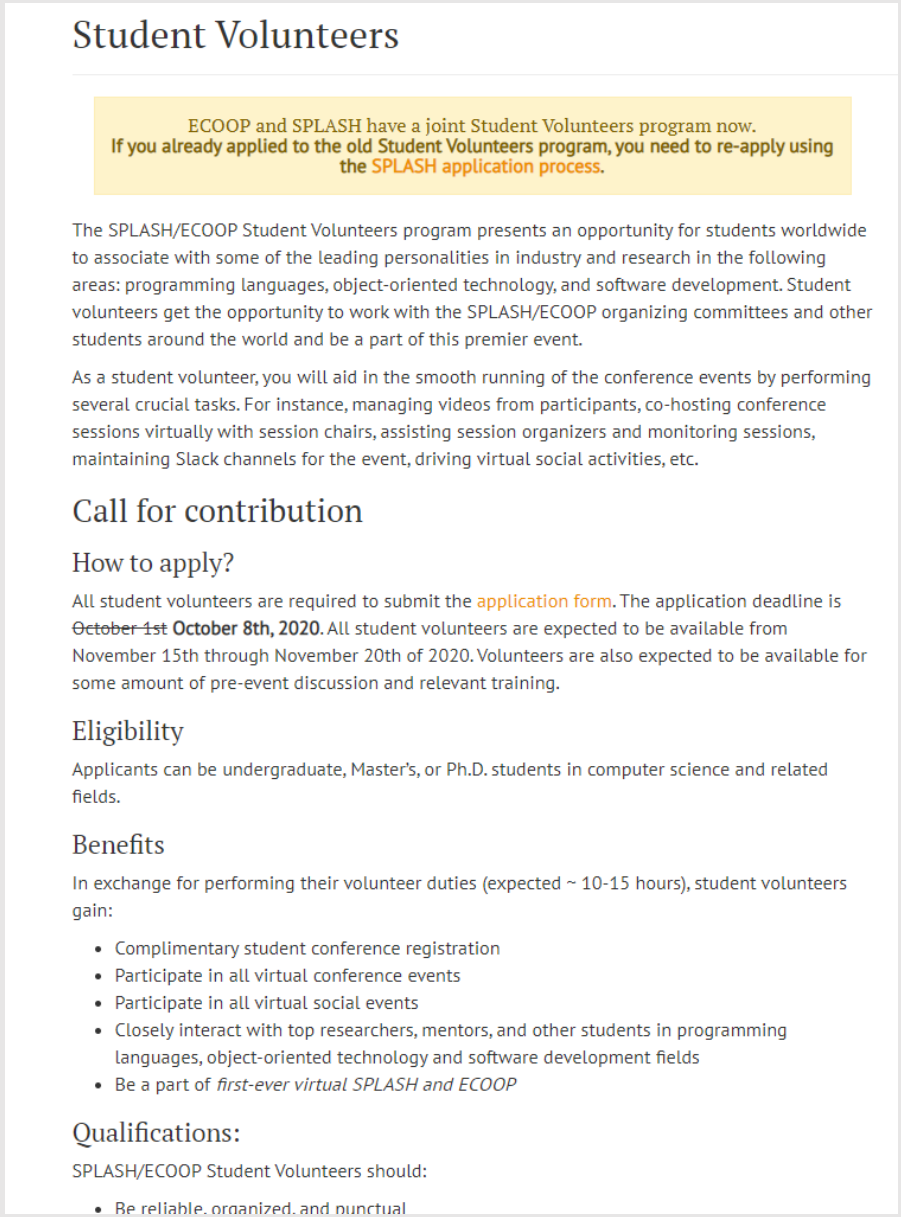
★	Title
☆	A Software Library Model for the Internet of Things Ian C. McCormack Link to publication
☆	Consolidation: A Technique for Improving Permissiveness of Human-Machine Interfaces Sang Heon Choi Link to publication
☆	Design and Implementation of a Gradual Verifier Mona Zhang, Jacob Gorenburg Link to publication File Attached
☆	Detecting Performance Patterns with Deep Learning Sophia Kolak



Alphabetical Listing of ACM SIGs

SIGs Listing

- SIGACCESS - Special Interest Group on Accessible Computing
- SIGACT - Special Interest Group on Algorithms & Computation Theory
- SIGAda - Special Interest Group on Ada Programming Language
- SIGAI - Special Interest Group on Artificial Intelligence
- SIGAPP - Special Interest Group on Applied Computing
- SIGARCH - Special Interest Group on Computer Architecture
- SIGBED - Special Interest Group on Embedded Systems
- SIGBio - Special Interest Group on Bioinformatics, Computational Biology
- SIGCAS - Special Interest Group on Computers and Society
- SIGCHI - Special Interest Group on Computer-Human Interaction
- SIGCOMM - Special Interest Group on Data Communication
- SIGCSE - Special Interest Group on Computer Science Education



Student Volunteers

ECOOP and SPLASH have a joint Student Volunteers program now.
If you already applied to the old Student Volunteers program, you need to re-apply using the SPLASH application process.

The SPLASH/ECOOP Student Volunteers program presents an opportunity for students worldwide to associate with some of the leading personalities in industry and research in the following areas: programming languages, object-oriented technology, and software development. Student volunteers get the opportunity to work with the SPLASH/ECOOP organizing committees and other students around the world and be a part of this premier event.

As a student volunteer, you will aid in the smooth running of the conference events by performing several crucial tasks. For instance, managing videos from participants, co-hosting conference sessions virtually with session chairs, assisting session organizers and monitoring sessions, maintaining Slack channels for the event, driving virtual social activities, etc.

Call for contribution

How to apply?

All student volunteers are required to submit the application form. The application deadline is ~~October 1st~~ **October 8th, 2020.** All student volunteers are expected to be available from November 15th through November 20th of 2020. Volunteers are also expected to be available for some amount of pre-event discussion and relevant training.

Eligibility

Applicants can be undergraduate, Master's, or Ph.D. students in computer science and related fields.

Benefits

In exchange for performing their volunteer duties (expected ~ 10-15 hours), student volunteers gain:

- Complimentary student conference registration
- Participate in all virtual conference events
- Participate in all virtual social events
- Closely interact with top researchers, mentors, and other students in programming languages, object-oriented technology and software development fields
- Be a part of *first-ever virtual SPLASH and ECOOP*

Qualifications:

SPLASH/ECOOP Student Volunteers should:

- Be reliable, organized, and punctual

Primer za 2020 godinu
https://2020.ecoop.org/

- direktni kontakt (organizacija)
- “student internship”
- twitter ~~instagram~~

The 1980 ACM Turing Award Lecture

Delivered at ACM '80, Nashville, Tennessee, October 27, 1980

C.A.R. Hoare

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960 to 1968 he held a number of positions with Elliot Brothers, Ltd., England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science *Honoris Causa* by the University of Southern California in 1979.

The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A. M. Turing, an English mathematician who made many important contributions to the computing sciences.

The Emperor's Old Clothes

Charles Antony Richard Hoare
Oxford University, England

The author recounts his experiences in the implementation, design, and standardization of computer programming languages, and issues a warning for the future.

Key Words and Phrases: programming languages, history of programming languages, lessons for the future
CR Categories: 1.2, 2.11, 4.2

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's present address: C. A. R. Hoare, 45 Banbury Road, Oxford OX2 6PE, England.

© 1981 ACM 0001-0782/81/0200-0075 $00.75.

My first and most pleasant duty in this lecture is to express my profound gratitude to the Association for Computing Machinery for the great honor which they have bestowed on me and for this opportunity to address you on a topic of my choice. What a difficult choice it is! My scientific achievements, so amply recognized by this award, have already been amply described in the scientific literature. Instead of repeating the abstruse technicalities of my trade, I would like to talk informally about myself, my personal experiences, my hopes and fears, my modest successes, and my rather less modest failures. I have learned more from my failures than can ever be revealed in the cold print of a scientific article and now I would like you to learn from them, too. Besides, failures

I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed

1972 ACM Turing Award Lecture

[Extract from the Turing Award Citation read by M.D. McIlroy, chairman of the ACM Turing Award Committee, at the presentation of this lecture on August 14, 1972, at the ACM Annual Conference in Boston.]

The working vocabulary of programmers everywhere is studded with words originated or forcefully promulgated by E.W. Dijkstra—display, deadly embrace, semaphore, go-to-less programming, structured programming. But his influence on programming is more pervasive than any glossary can possibly indicate. The precious gift that this Turing Award acknowledges is Dijkstra's *style*: his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design. He has published about a dozen papers, both technical and reflective, among which are especially to be noted his philosophical addresses at IFIP,[1] his already classic papers on cooperating sequential processes,[2] and his memorable indictment of the go-to statement.[3] An influential series of letters by Dijkstra have recently surfaced as a polished monograph on the art of composing programs.[4]

We have come to value good programs in much the same way as we value good literature. And at the center of this movement, creating and reflecting patterns no less beautiful than useful, stands E.W. Dijkstra.

The Humble Programmer

by Edsger W. Dijkstra

As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952, and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing is the slowness with which, at least in my part of the world, the programming profession emerged, a slowness which is now hard to believe. But I am grateful for two vivid recollections from that period that establish that slowness beyond any doubt.

After having programmed for some three years, I had a discussion with van Wijngaarden, who was then my boss at the Mathematical Centre in Amsterdam—a discussion for which I shall remain grateful to him as long as I live. The point was that I was supposed to study theoretical physics at the University of Leiden simultaneously, and as I found the two activities harder and harder to combine, I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to a formal completion only, with a minimum of effort, and to become . . . , yes what? A programmer? But was that a respectable profession? After all, what was programming? Where was the sound body of knowledge that could sup-

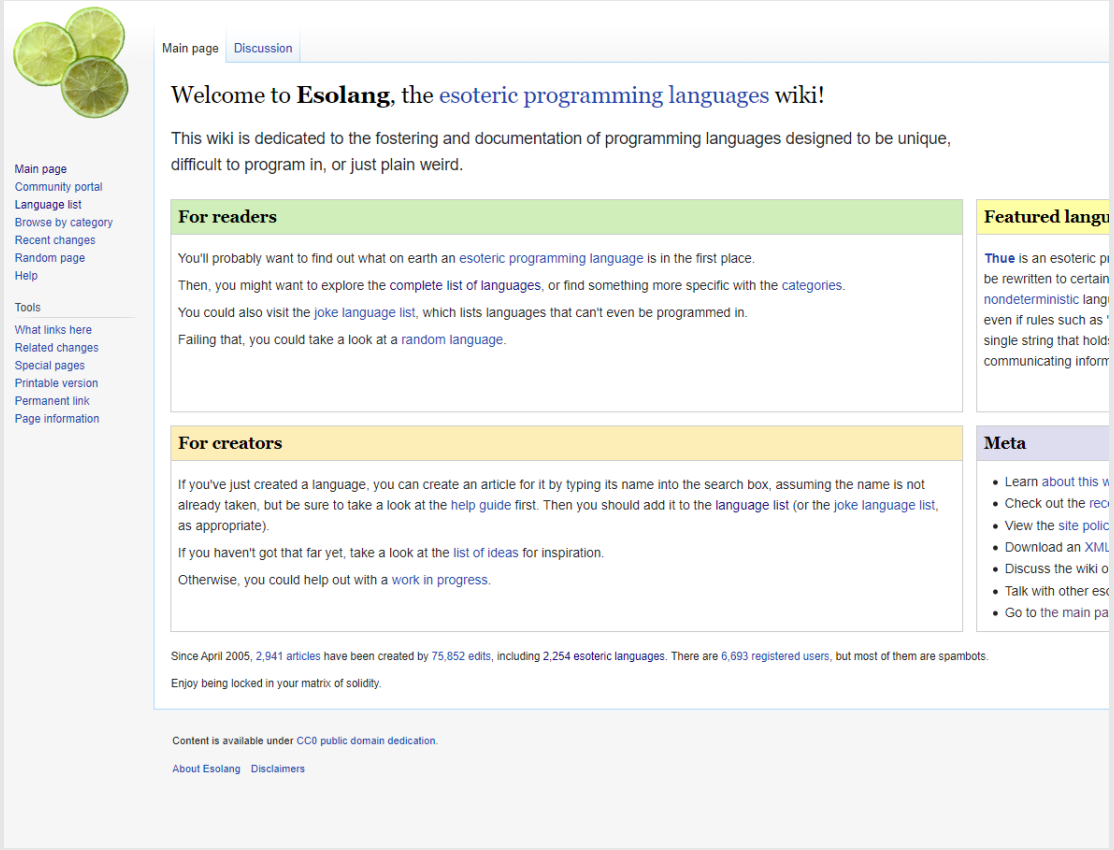
Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

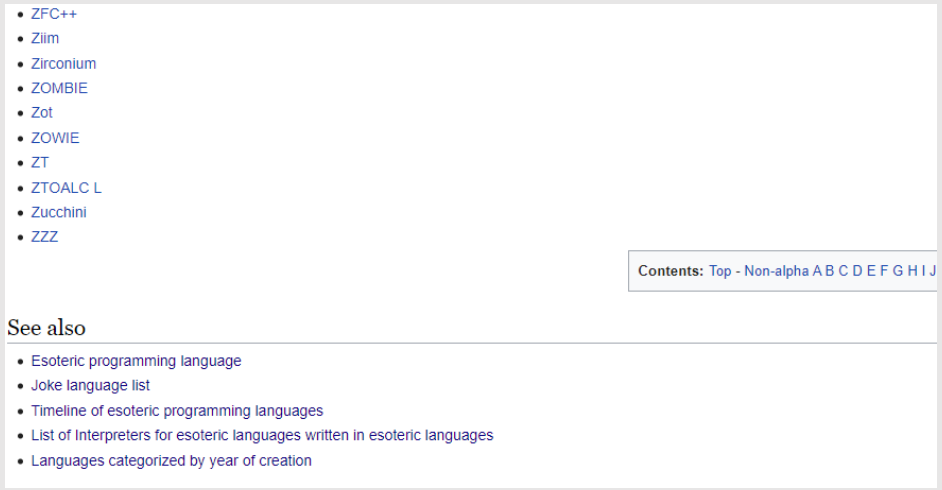
1,2,3,4 Footnotes are on page 866.

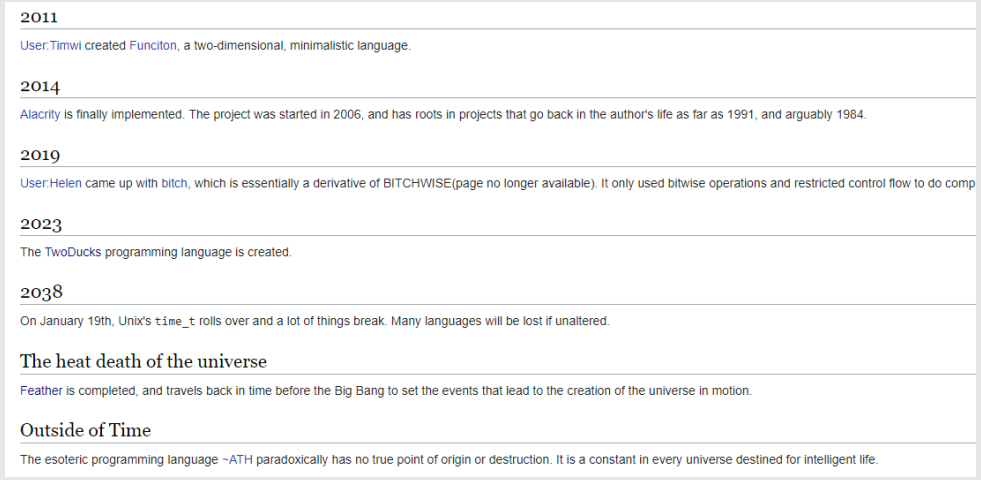
Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

https://www.cs.utexas.edu/users/EWD/ (za početak: EWD316)

https://esolangs.org/wiki/Main_Page







`mov` is Turing-complete

Stephen Dolan
Computer Laboratory, University of Cambridge
stephen.dolan@cl.cam.ac.uk

Abstract

It is well-known that the x86 instruction set is baroque, overcomplicated, and redundantly redundant. We show just how much fluff it has by demonstrating that it remains Turing-complete when reduced to just one instruction.

The instruction we choose is `mov`, which can do both loads and stores. We use no unusual addressing modes, self-modifying code, or runtime code generation. Using just this instruction (and a single unconditional branch at the end of the program to make nontermination possible), we demonstrate how an arbitrary Turing machine can be simulated.

1. Introduction

The `mov` instruction on x86 has quite a few addressing modes. This instruction can be used to do memory loads or stores, as well as loading immediate constants into registers. Powerful as it is, it doesn't do any form of conditional branching or comparison, so it's not obvious that it is Turing-complete on its own.

Of course, on an actual x86 processor the `mov` instruction can be used to write arbitrary code into memory after the instruction pointer which the processor will then execute, making it in some sense trivially "Turing-complete". We consider this cheating: our simulating of a Turing machine uses no self-modifying code nor runtime code generation, and uses no obscure addressing modes. In fact, the addressing modes used are available as instructions on most RISC architectures, although RISC machines generally don't call them all `mov`.

Executing a finite sequence of `mov` instructions will complete in a finite amount of time. In order to have Turing-completeness, we must allow for nontermination. So, our Turing machine simulator consists of a sequence of `mov` instructions, followed by an unconditional branch back to the start.

2. Machine model

We work with a simple abstract machine model. Our machine has a random access memory composed of words. Each word can hold either a memory address or an offset, where offsets are either 0 or 1 (which are not valid memory addresses). We have n registers $R_1, \ldots, R_n$, which also hold a word each. We assume plenty of registers for now, but later we show how their number can be reduced without losing expressiveness.

We have the following instructions (if you like RISC) or addressing modes (if you like CISC). We use Intel x86 syntax, where the `mov` instructions have the destination first and the source second, and square brackets indicate memory operands.

Instruction	x86 syntax
Load Immediate	`mov` R_{dest}`, c`
Load Indexed	`mov` R_{dest}`, [`R_{src} `+` R_{offset}`]`
Store Indexed	`mov [`R_{dest} `+` R_{offset}`],` R_{src}

On x86, these are all addressing modes of the same `mov` instruction. However, even on RISC machines these are still single instructions. For instance, on PowerPC these three operations are `li`, `ldx` and `stx`.

It would appear that we are cheating slightly by allowing arithmetic in addresses. However, our "arithmetic" is of a very restricted form: the indexed instructions may only be used when R_{src} (or R_{dest} for stores) is an even-numbered memory addresses. Since offsets are always 0 or 1, then our "arithmetic" can be implemented as bitwise OR, or even bitstring concatenation.

From these three instructions, we have a few simple derived instructions. We can use the load indexed and store indexed instructions with a constant offset by using a temporary register. For instance, the load instruction `mov` R_{dest}`, [`R_{src}`]` can be implemented as follows, using the register `X` as a temporary:

```
mov X, 0
mov Rdest, [Rsrc]
```

As it happens, these constant-offset instructions are available as other addressing modes of `mov` on x86.

Memory is logically divided into cells, which are pairs of adjacent words which start on even-numbered memory addresses. Our load indexed and store indexed operations can be viewed as load and store to a cell, where the address of the cell is given by one register, and which of the two words to access is specified by another register.

Using just these instructions, we can simulate an arbitrary Turing machine.

3. Representing Turing machines

A Turing machine $\mathcal{M}$ is a tuple

$$\mathcal{M} = (Q, q_0, \Sigma, \sigma_0, \delta)$$

whose components are:

- A finite set of states Q, with a distinguished start state $q_0 \in Q$.
- A finite set of symbols Σ, with a distinguished blank symbol $\sigma_0 \in \Sigma$.
- A transition table δ, which is a partial function $Q \times \Sigma \to \Sigma \times \{L, R\} \times Q$.

[Copyright notice will appear here once 'preprint' option is removed.]

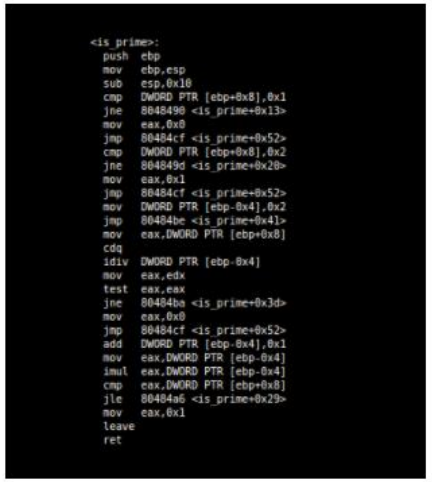
https://github.com/xoreaxeaxeax/movfuscator

Overview

The M/o/Vfuscator (short 'o', sounds like "mobfuscator") compiles programs into "mov" instructions, and only "mov" instructions. Arithmetic, comparisons, jumps, function calls, and everything else a program needs are all performed through mov operations; there is no self-modifying code, no transport-triggered calculation, and no other form of non-mov cheating.

The basic effects of the process can be seen in overview, which illustates compiling a simple prime number function with gcc and the M/o/Vfuscator.

Assembly:

GCC	M/o/Vfuscator
	

https://youtu.be/R7EEoWg6Ekk

- Programski jezici – sintaksa je samo okvir
- Programski jezici – ogromna oblast
- Teorija je više prisutna nego što smo mi svesni toga
- Problemi koje imamo su verovatno već rešeni, ali možda tražimo rešenje na pogrešnom mestu
- Ne optimizuj, bar ne prerano, a posebno ne napamet
- Uvek dobra ideja: više radoznalosti, manje površnosti

Želim vam puno zabave, izazova i uspeha u korišćenju i istraživanju, a možda i razvoju programskih jezika