Gymnázium Arabská, Praha 6, Arabská 14

Obor programování

Programovací jazyk Slang

Adam Suchý

Duben, 2024

Zdrojový kód je veřejně dostupný pod licencí MIT. Její plné znění v anglickém jazyce je přiloženo v souboru `LICENSE` v repozitáři projektu.

dále:

Prohlašuji, že jsem jediným autorem tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V ..... dne ..... Adam Suchý .....

Anotace

Slang je víceúčelový programovací jazyk založený na principu datových proudů. V této práci jsem definoval jeho syntaxi a naprogramoval překladač do LLVM IR. Dále vysvětluji postup, jak překladač funguje a jak by vývoj Slangu mohl pokračovat.

Abstract

Slang is a general purpose programming language with first-class support for data streams. In this work I define it's syntax and build a compiler into LLVM IR. Additionally, I explain how the compiler works and how its future developement could look like.

Obsah

1. Úvod **2**

1.1. Původní znění zadání ... 2

1.2. Použité technologie ... 2

2. Definice jazyka **2**

2.1. Datové typy a proměnné ... 3

2.2. Kolony ... 3

2.3. Operátory ... 4

2.4. Vstupní bod ... 4

2.5. Řídící struktury ... 5

2.6. Volání externích funkcí ... 5

3. Překladač **5**

3.1. Lexer a parser ... 5

3.2. Statická analýza ... 6

3.3. Generování kódu ... 7

3.3.1. Struktura LLVM kódu streamu ... 7

3.3.2. Volání kolon ... 8

3.3.3. Runtime ... 8

4. Podněty k dalšímu vývoji **9**

5. Instalace **9**

6. Závěr **9**

Odkazy **10**

1. Úvod

V rámci této práce jsem vytvořil překladač pro vlastní víceúčelový programovací jazyk, který jsem pojmenoval Slang. Základní myšlenka jazyka je, že většina programů zpracovává datové proudy: různými způsoby s nimi manipuluje a upravuje je. V případě, že dat je velké množství, si často musí programátor psát vlastní funkce, aby nemusel všechna data držet v paměti najednou. Pro populární programovací jazyky již samozřejmě existují knihovny, které práci s datovými proudy ulehčují. Mě ale zajímalo, jak by mohl vypadat jazyk, který má tuto funkcionalitu zabudovanou do jeho samotné syntaxe.

1.1. Původní znění zadání

Slang bude turingovsky kompletní programovací jazyk založený na principu datových proudů. Cílem jazyka je prozkoumat možnost programování aplikací jako program, co vytváří, mění a zpracovává datové proudy. Toto může mít několik výhod, hlavně v možnosti paralelizace jednotlivých komponent.

Součástí jazyka bude:

1. silný typový systém,
2. vestavěná podpora principu datových proudů,
3. možnost využití paradigmatu funkcionálního programování (funkce a datové proudy lze spojovat a pracovat s nimi jako s hodnotami).

Součástí projektu bude samotná specifikace jazyka a k tomu přiložený LLVM frontend (překladač do LLVM Itermediate Representation).

1.2. Použité technologie

Pro kód překladače jsem zvolil čistě funkcionální programovací jazyk Haskell. [1] Tuto volbu jsem udělal, protože celý překladač je v zásadě matematická funkce, která operuje nad textem. Haskell dokáže velmi jednoduše skládat funkce dohromady a vytvářet z nich komplikovanější struktury. Zároveň pro Haskell existuje mnoho knihoven například pro parsování.[1] Ostatní programovací jazyky většinou závisí na nástrojích, který parser vygenerují.

Překladač Slangu také negeneruje strojový kód, ale je jen frontend pro LLVM - generuje LLVM IR (Intermediate Representation). LLVM IR je jazyk velmi blízký assembly, ale není specifický pro architekturu procesoru [2]. Překladač do LLVM IR tak nemusí brát v potaz různé architektury.

LLVM má poté svůj vlastní překladač, který dokáže IR optimalizovat a přeložit do strojového kódu. Vygenerované LLVM IR může tudíž být pomalé s instrukcemi navíc, protože konečný program je optimalizovaný LLVM překladačem.

Další LLVM frontendy jsou například `rustc` (překladač jazyka Rust), nebo `clang` (překladač jazyků C a C++). Jako poslední krok při překladu Slangu se tedy spouští LLVM překladač, který zároveň provede optimalizaci kódu.

2. Definice jazyka

Základní stavební jednotkou Slangu jsou transformátory datových proudů (dále označovány jako **streamy**). Definují se pomocí klíčového slova `stream` na úrovni souboru a mají jeden vstupní a výstupní typ.

[1] Parsování: viz Kapitola 3.1

Celý kód ve streamu je nekonečná smyčka, která se dá přerušit posláním hodnoty do výstupu. Pokud bude ale vyžádána další výstupní hodnota, pak se smyčka opět spustí tam, kde naposledy skončila. Všechny deklarované proměnné budou pořád existovat.

```
stream proud i32 -> i32 { }
```

Kód 1: Definice streamu

2.1. Datové typy a proměnné

Ve Slangu jsou všechny objekty typované. Je tedy nutné znát jejich typ už při jejich deklaraci a ten nelze měnit. Proměnné je ale možné deklarovat znovu jako jiný typ, ale po deklaraci ztratí svou původní hodnotu a v paměti mají nové místo.

Proměnné jsou deklarovány pomocí klíčového slova `let`. Při deklaraci je možné upřesnit typ, ale není potřeba. Když není typ specifikován, tak se odvozen od nastavené hodnoty. Pokud se proměnná pouze deklaruje, ale není rovnou inicializována hodnotou, je nutné typ specifikovat.

Základní datové typy jsou různé reprezentace čísel se znaménkem, kterým LLVM rozumí. Pro Slang to jsou:

`bool`, `char`, `i32`, `i64`, `float`, (`void`, `...`)

Typy `void` a `...` slouží pouze ke kompatibilitě s externími funkcemi napsaných v jiných programovacích jazycích.

Z typů je možné skládat uspořádané n-tice tím, že je dáme společně do závorek. Interně se těmto typům říká `tuple`. Z n-tice je možné jednotlivé prvky vybrat pomocí číselných atribut. Pokud má proměnná `x` typ `(i32, bool)`, pak `x.0` bude mít typ `i32`.

Číselné konstanty jsou vždy typu `i64`. Ke každému typu také existuje ukazatel, který ukazuje na místo v paměti, kde je uložený právě ten typ (například `&i64` je ukazatel na `i64`).

```
let b: bool = false;

// Typ není potřeba specifikovat. Typ proměnné x bude i64.
let x = 0;

let y: &i32; // Proměnná nemá hodnotu - není inicializována

let t = (true, '\b');
let prvni: bool = t.0;
let druhy: &char = &t.1;
```

Kód 2: Deklarace proměnných

2.2. Kolony

Důležitou funkcí Slangu jsou kolony streamů. Každá kolona se skládá z jedné vstupní hodnoty a několika streamů. Speciální kolony `in` a `out` značí vstup a výstup aktuálního streamu.

Do *námi* vytvořených kolon je možné poslat **pouze jednu** vstupní hodnotu hned při jejich vytvoření, ale do kolony `out` můžeme posílat hodnoty vždy. Hodnota poslaná do `out` pak bude dostupná v následujícím streamu v koloně.

Z kolon můžeme hodnoty také vybírat (chytat) pomocí klíčového slova `catch`. Jestliže použijeme `catch` na kolonu, která nemá další výstup (je ukončená), pak se aktuální stream a s ním i celá aktuální kolona ukončí.

Vytvořené kolony lze uložit do proměnné, ale tu nelze přepsat, ani použít jinak, než při chytání výstupu kolony (viz Kapitola 3.3.2).

Vytvoření kolony, která zčtyřnásobí svůj vstup:

```
stream double i64 -> i64 {
  (catch in) * 2 | out;
}

stream ukazka () -> () {
  let kolona = 2 | double | double;
  let ctyri: i64 = catch kolona; // 4
}
```

Kód 3: Kolona pro čtyřnásobek

2.3. Operátory

Slang rozumí následujícím operátorům:

1. Matematické operátory: `*`, `/`, `+`, `-`. Definované pro všechny číselné typy.
2. Porovnávací operátory: `==`, `!=`, `<`, `>`, `<=`, `>=`. Definované pro všechny číselné typy.
3. Logické operátory: `!`, `&&`, `||`. Definované pro typ `bool`.
4. Operátory reference a dereference: `&`, `*`. Definované pro hodnoty v proměnných a ukazatele.
6. Indexování: `[idx]`. Definované pro ukazatele.
6. Operátory kolon: `catch`, `|`
7. Přetypovací operátor: `as`.

   Přetypovat lze mezi sebou pouze celočíselné typy a ukazatele.

Priorita operací (sestupně):
`[idx]`, `(&, *)`, `as`, `(*, /)`, `(+, -)`, `(==, !=, <, >, <=, >=)`, `(!, &&, ||)`, `|`, `(catch, =)`

```
let str: &char = "Hello World!";
let prvniZnak = *str; // 'H'
let druhyZnak = *str[1] as i32; // číselná hodnota 'e' s typem i32
```

Kód 4: Operátory

2.4. Vstupní bod

Vstupním bodem programu je stream `main`, který dostane známou dvojici `(i32, &&char)` reprezentující počet argumentů a ukazatel do pole řetězců s argumenty. Zároveň musí mít návratový typ `i32` a vracet standardní chybný kód.

```
// vstupní bod programu
stream main (i32, &&char) -> i32 {
  0 as i32 | out;
}
```

Kód 5: Vstupní bod programu

2.5. Řídící struktury

Slang podporuje dvě základní řídící struktury: `if` a `while`.

```
let sum = 0;
let i = 0;
while sum < 100 {
  sum = sum + (i = i + 1);
}
if (i > 5) {
  "vetsi nez 5, mozna i 6, nebo 7, co ja vim" | out;
  if i > 6 {
    "vetsi nez 6" | out;
  } else if i > 7 {
    "vetsi nez 7" | out;
  }
} else {
  "mensi nebo rovno 5" | out;
}
```

Kód 6: Řídící struktury if a while

2.6. Volání externích funkcí

Pro případy, kdy je lepší napsat funkci v jiném jazyce, nebo když je potřeba interagovat se systémem, je možné zavolat funkce z externích knihoven. Nejdříve je ale nutné funkci deklarovat jako externí, aby Slang věděl, že existuje. Například deklarace známé funkce `printf` vypadá takto:

```
extern fn printf (&char, ...) -> i32;
```

Kód 7: Deklarace externí funkce

V kódu 7 můžeme vidět použití typu `...`, který značí, že funkce má dynamický počet argumentů.

Před spuštěním programu je nutné zkompilovaný objekt spojit se statickou knihovnou, nebo deklarovat dynamický linker, pomocí libovolného linkeru. Přiložený skript `sc` používá LLVM linker `lld`.

3. Překladač

Překladače jsou většinou rozděleny na několik částí, které na sebe navazují. Postupně text zpracuje lexer, parser, statický analyzér a nakonec se generuje výsledný kód. [3] V případě Slangu se jedná o LLVM IR.

Překladač Slangu má spojené fáze lexování a parsování a také analýzu s generováním kódu.

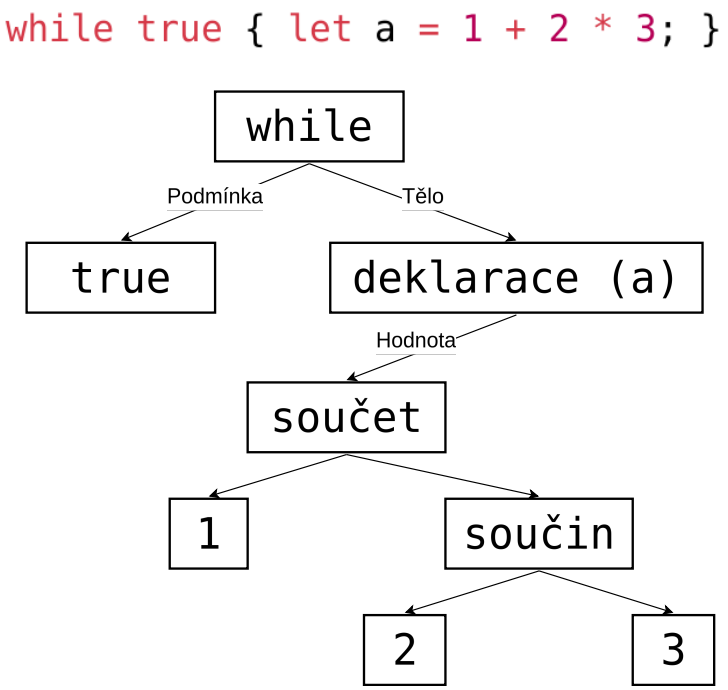
3.1. Lexer a parser

Překladače v první fázi provádí tokenizaci (lexikální analýzu) textu. Převádí tím prostý text na syntakticky významné tokeny (lexémy). Zjednodušují tím práci pro parser, který se nemusí tím pádem zabývat mezerami, nebo novými řádky, ale může soubor chápat jako seznam klíčových slov, identifikátorů apod.

Pro Haskell existuje již několik knihoven, které buď přímo provádí lexikální analýzu, nebo generují lexer. Například nástroj `alex` [4] dokáže z definice tokenů jazyka ve svém vlastním

formátu vygenerovat velmi rychlý lexer s využitím deteriministických konečných automat. [5] Největší jejich výhodou je složitost s jakou dokáží identifikovat lexémy v textu, která je lineární ($O(n)$) vzhledem k délce vstupních dat. [6], [7]

Parser následně z tokenů skládá syntaktický strom (AST), který reprezentuje operace a jejich závislosti na jiných operacích (viz. Obrázek 1). Celá struktura programu je v AST zaznamenána a například interpretované jazyky už mohou s tímto přímo vykonat program.



Obrázek 1: Syntaktický strom while smyčky s deklarací proměnné

Pro psaní parserů existuje také mnoho knihoven. Například `happy` je podobný nástroj, jako `alex` (také od stejného vývojáře) a generuje optimalizované parsery z gramatiky v Backus-Naurově normální formě. Happy umí vygenerovat LALR a GLR parsery. [8] Oba typy parserů pochází z rodiny LR (**L**eft to Right, **R**ightmost derivation) parserů.

Slang ale nepoužívá `alex`, ani `happy`. `alex` byl dobrý kandidát pro lexikální analýzu Slangu kvůli jeho rychlosti. Nakonec byla ale jeho integrace se zvolenou knihovnou pro parsování moc složitá.

Z textu generuje AST rovnou ručně napsaný parser pomocí knihovny Megaparsec. Princip Megaparsec knihovny je skládání menších parserů do větších. [9] Například parser pro součet je parser, který nejdřív načte levého sčítance, pak token s významem součtu a nakonec pravého sčítance. Parser má tedy strukturu rekurzivního sestupu.

Výhodou tohoto systému je možnost vytvoření si jednoduchých parserů pro tokeny rovnou z textu, takže tokenizaci provádí samotný parser, který tím pádem ví, kde nastala chyba, a může vytvářet lepší chybové hlášky. Když jsou tyto kroky rozdělené, tak je mnohem obtížnější si uchovat původní lokaci tokenů ve zdrojovém kódu.

3.2. Statická analýza

Dalším krokem v překladu zdrojového kódu bývá statická analýza syntaktického stromu. Jde především o kontrolu, že operace dávají smysl, že existují pro dané typy a že existují volané funkce, nebo použité proměnné. Často se ke všem vrcholům stromu také přidá typ výsledku operace. [10]

Překladač Slangu statickou analýzu provádí rovnou při generování kódu. [11] Kód překladače je pak mnohem kratší, protože se neduplikuje struktura jednotlivých vrcholů stromu a zároveň se lépe dohledávají chyby. Nesměl ji ale také vynechat, protože u všech LLVM instrukcí je nutné specifikovat typ argumentů. To pomáhá zejména při optimalizaci a případném ladění. [12]

3.3. Generování kódu

Zdaleka nejtěžší část je konverze AST do instrukcí. Naštěstí jsem mohl využít LLVM, takže jsem nemusel dohledávat rozdíly mezi architekturami procesorů, nebo generovat validní binární aplikace.

Vstup pro generátor kódu je několik stromů, každý reprezentující jeden stream. Generátor ze všech definicí streamů a deklarací funkcí extrahuje jejich typy argumentů a návratových hodnot a vytvoří globální seznam identifikátorů s odkazy na jednotlivé funkce. Instrukce pro každý stream jsou pak generovány kompletně odděleně, jenom s kontextem globálních funkcí.

```
@str2 = private constant [14 x i8] c"Hello world!\0A\00", align 1

%stream_main_locals = type {}
define i1 @stream_main(ptr %l, ptr %rp, %clt* %cl, i8** %b) noinline {
  %1 = load ptr, ptr %b
  indirectbr ptr %1, [ label %.block0, label %.block1, label %.blockblocked ]

.block0:
  %2 = call i32 (ptr, ...) @printf(ptr @str2)
  %3 = trunc i64 0 to i32
  store i32 %3, ptr %rp
  store i8* blockaddress(@stream_main, %.block1), i8** %b
  ret i1 1

.block1:
  br label %.block0

.blockblocked:
  store i8* blockaddress(@stream_main, %.blockblocked), i8** %b
  ret i1 0
}
```

LLVM kód streamu `main`, který vypíše „Hello World!“

3.3.1. Struktura LLVM kódu streamu

Vzhledem k tomu, že streamy nejsou jednoduché funkce, tak se také nemohou stejně volat. Musí si vždy pamatovat své lokální proměnné i po tom, co se jako funkce ukončí. Všechna paměť alokovaná pomocí instrukce `alloca` je po návratu funkce smazána[2] a je tedy nutné, aby funkci byla alokovaná paměť předána už při jejím volání. Všechna paměť je tím pádem alokována hned při startu programu ve skutečné *funkci* `main` (nikoliv *streamu*, viz Kapitola 3.3.3). Každý stream si také musí pamatovat místo, kde naposledy skončil, aby poté mohl pokračovat.

Při generování streamu je tedy potřeba také vygenerovat typ se všemi jeho lokálními proměnnými. V ukázce LLVM kódu se jedná o typ `%stream_main_locals`[3], který dostane funkce

[2]Paměť není *skutečně* smazána, ale je volná k použití, takže ji může jakákoliv jiná funkce přepsat. Vzhledem k tomu, že je paměť mimo přímou kontrolu funkce, která jí alokovala, můžeme jí považovat za smazanou.

[3]Ten je prázdný, protože stream nemá žádné lokální proměnné.

jako první parametr. Blok, kde má stream pokračovat, se ukládá do speciální proměnné (v ukázce `i8** %b`).

3.3.2. Volání kolon

Při volání kolony se streamy spouští od konce. Výstup z kolony je poté jednoduše přístupný, ale bohužel se komplikuje předání vstupní hodnoty.

Nejdříve je potřeba znát z jakých streamů se kolona skládá a jak jdou po sobě. Pro inicializaci kolony se v kódu musí vygenerovat konstanta s ukazateli na všechny funkce v koloně. Také se vytvoří speciální typ, který v má v sobě seřazené všechny typy lokálních proměnných streamů. Alokace lokálních proměnných se tedy děje při inicializaci celé kolony. Streamy si pak jen předávají ukazatele na část alokované paměti s jejich proměnnými.

```
@pipeline_1 = internal constant [3 x %clt*] [
  %clt* @stream_double,
  %clt* @stream_double,
  %clt* @const_copy
]
; %sptr a i64 jsou vstupem pro @const_copy
%pipeline_1_stack = type {%stream_double_locals,
  %stream_double_locals, %sptr, i64}

; kombinovaný typ s lok. prom. kolony a místem pro ukazatele
; bloků, ve kterých mají streamy pokračovat
%pipeline_1_stack_comb = type { %pipeline_1_stack, [3 x i8*] }
```

LLVM kód kolony z ukázky 3

Aby mohla kolona dostat vstup, je na konec přidán (ve zdrojovém kódu technicky na začátek) speciální stream `@const_copy` (viz Kapitola 3.3.3), který chápe lokální proměnné jako vstup a jen ho překopíruje do svého výstupu. Poté se hned zablokuje. Žádný stream tedy není v koloně poslední a jejich kód může být vždy stejný. Svůj vstup buď dostává z jiného streamu, nebo z `@const_copy`.

Vrácená hodnota z funkce streamu je boolean, který značí, jestli volaný stream vrátil hodnotu (jestli nebyl zablokovaný). Podle toho se volající stream rozhodne, jestli bude pokračovat dál, nebo jestli se také ukončí.

3.3.3. Runtime

Aby se přeložený program mohl spustit, musí se zavolat funkce `main`.[4] Vstupní bod programu napsaného ve Slangu je ale stream a tím pádem je potřeba k vygenerovanému kódu funkci přidat. Runtime se tedy stará o inicializaci a poté finální ukončení programu. Alokuje všechnu paměť, spustí hlavní kolonu (skládající se ze streamu `main` a `@const_copy`) a předá hodnoty `argc` a `argv`. Nakonec vyzvedne výstupní hodnotu `main` a vrátí ji jako chybný kód.

V runtime se také nachází stream `@const_copy`, protože je potřeba vždy už při inicializaci. Runtime zůstává vždy pro všechny programy stejný.

[4] Ještě před zavoláním main funkce se spouští inicializační kód, který pro Slang definují knihovny `/lib/crt1.o` a `/lib/crti.o`.

4. Podněty k dalšímu vývoji

Do Slangu bych určitě rád přidal ještě několik funkcí. Většinou se spíše jedná o zjednodušení něčeho, co už ve Slangu udělat lze. Myslím, že kritické funkce už Slang umí.

Například by bylo dobré přidat známý příkaz `break`, jednak pro brzké ukončení smyčky, ale také pro ukončení streamu. Po příkazu by se stream hned zablokoval. K lepší logice zablokování by také mělo patřit podmíněné chytání z kolon například pomocí `if let`, nebo `while let`.

```
stream jedna () -> i64 {
  1 | out;
  break;
}

stream ukazka () -> () {
  let kolona = () | jedna;
  let jedna = catch kolona;

  if let dalsi = catch kolona {
    printf("dostal jsem 1 dvakrát\n");
  }

  () | out;
}
```

Kód 8: Lepší logika zablokování streamů

Určitě by také bylo dobré mít podporu pro definované struktury složených z více typů. Toto není nezbytná funkce, jelikož Slang podporuje anonymní n-tice, ale pojmenované struktury by pro vývojáře byly určitě ergonomičtější.

Další nedostatek je separátní proměnná pro pamatování si místa, kde stream skončil. Bylo by lepší si tuto informaci uchovat rovnou vedle lokálních proměnných. Inicializace kolon by také byla o něco rychlejší.

Jako jednu z výhod streamů ve Slangu jsem bral jejich oddělenost a možnost paralelizace. S aktuálně generovaným LLVM kódem se kolony rozhodně paralelizovat nedají a aby se dali, musel by se kompletně změnit systém jejich volání.

5. Instalace

Instalace překladače a jeho spuštění je popsáno v souboru `README.md` v repozitáři projektu.

6. Závěr

Úspěšně jsem vytvořil překladač pro vlastní jazyk Slang a myslím, že jsem tím pádem zadání splnil. I přes to má Slang jasné cesty, kudy by se mohl dál vyvíjet. V průběhu práce jsem se naučil jak fungují překladače a jak vypadá program na úrovni instrukcí. Programování v Haskellu bylo také velmi zajímavé a naučilo mě operovat v paradigmatu funkcionálního programování.

Seznam ukázek kódu v jazyce Slang

Kód 1: Definice streamu 3
Kód 2: Deklarace proměnných 3
Kód 3: Kolona pro čtyřnásobek 4
Kód 4: Operátory 4
Kód 5: Vstupní bod programu 4
Kód 6: Řídící struktury if a while 5
Kód 7: Deklarace externí funkce 5
Kód 8: Lepší logika zablokování streamů 9

Odkazy

[1] Webové stránky Haskell.org. [Online]. Dostupné z: https://haskell.org/

[2] C. Lattner a V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, bře. 2004.

[3] K. Schwarz, Přednáška 00: Intro to Compilers. [Online]. Dostupné z: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/00/Slides00.pdf

[4] S. Marlow a vývojáři Alex, Dokumentace nástroje Alex. [Online]. Dostupné z: https://haskell-alex.readthedocs.io/en/latest/introduction.html

[5] S. Marlow a vývojáři Alex, Zdrojový kód nástroje Alex. [Online]. Dostupné z: https://github.com/haskell/alex/blob/master/src/DFA.hs

[6] Wikipedia contributors, Automata theory — Wikipedia, The Free Encyclopedia. [Online]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Automata_theory&oldid=1215199334

[7] K. Schwarz, Přednáška 01: Lexical Analysis. [Online]. Dostupné z: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/01/Slides01.pdf

[8] S. Marlow a vývojáři Happy, Dokumentace nástroje Happy. [Online]. Dostupné z: https://haskell-happy.readthedocs.io/en/latest/introduction.html

[9] Megaparsec vývojáři, Paolo Martini a Daan Leijen, Stránka knihovny Megaparsec. [Online]. Dostupné z: https://hackage.haskell.org/package/megaparsec

[10] J. Morag, Micro C, Part 2: Semantic Analysis. [Online]. Dostupné z: https://blog.josephmorag.com/posts/mcc2/

[11] A. Suchý, Tato myšlenka mě napadla ve snu. 2024.

[12] LLVM vývojáři, LLVM Language Reference Manual. [Online]. Dostupné z: https://llvm.org/docs/LangRef.html#type-system