FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

mlang

KIV/FJP – Semestrální Práce

Jméno a přijmení: Martin Forejt
Osobní číslo: A20N0079P

Datum: 16. 12. 2020

1 Zadání

Cílem práce bude vytvoření překladače zvoleného jazyka. Je možné inspirovat se jazykem PL/0, vybrat si podmnožinu nějakého existujícího jazyka nebo si navrhnout jazyk zcela vlastní. Dále je také potřeba zvolit si pro jakou architekturu bude jazyk překládán (doporučeny jsou instrukce PL/0, ale je možné zvolit jakoukoliv instrukční sadu pro kterou budete mít interpret - nepište vlastní interpret, jednou z podúloh je vypořádat se s omezeními danými zvolenou platformou).

Jazyk musí mít minimálně následující konstrukce:

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- základní aritmetiku a logiku (+, -, *, /, AND, OR, negace a závorky, operátory pro porovnání čísel)
- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání

2 Úvod

Cílem této práce je návrh vlastního programovacího jazyka (mlang) a tvorba jeho překladače s výstupem pro reálnou platformu (.exe soubor pro Windows). Kromě minimálních požadavků (viz zadání) jsou požadavky na jazyk následující:

- Silně typovaný se statickou typovou kontrolou
- Datové typy integer, boolean, double, char, string
- Dynamicky alokovaná paměť (pro pole)
- Libovolný kód lze psát i mimo funkce

3 Nástroje

Překladač je napsán v programovacím jazyce C++, pro automatizaci překladu je použit CMake.

3.1 Flex

Pro lexikální analýzu je použit Flex – nástroj, který generuje zdrojový kód pro lexikální analyzátor. Jde o GNU variantu programu Lex. Používá se často spolu s generátorem syntaktického analyzátoru yacc nebo jeho vylepšenou alternativou GNU bison. [4]

3.2 Bison

GNU Bison je generátor syntaktického analyzátoru, který kontroluje syntaxi programu a pomáhá vytvářet AST[1].

3.3 LLVM

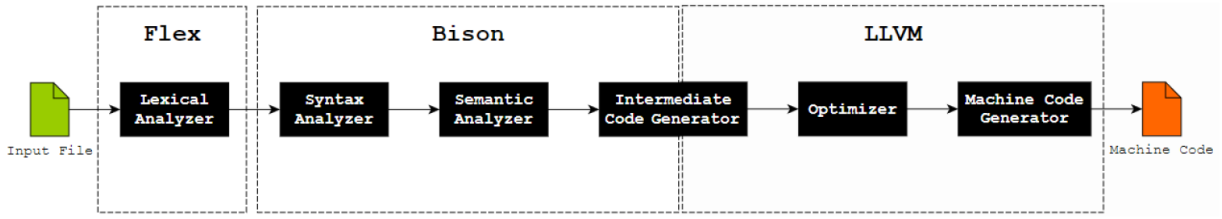
Jelikož cílem práce je tvorba překladače pro reálnou platformu (.exe soubor pro Windows), hledal jsem nějaký nástroj, který by toto usnadnil. Objevil jsem LLVM projekt [1] (který jsem předtím

[1] Abstraktní syntaktický strom

neznal) a po přečtení oficiálního návodu [2], jak pomocí LLVM vytvořit vlastní překladač, jsem se rozhodl použít LLVM i v této práci.

Postup při kompilaci bude tedy následovný:

1. Pomocí Flexu převedeme zdrojový kód na jednotlivé tokeny
2. Pomocí Bisonu provedeme syntaktickou analýzu a vygenerujeme AST
3. Procházíme AST a generujeme mezikód (Intermediate Code, LLVM IR)
4. Pomocí LLVM mezikód optimalizujeme
5. Pomocí LLVM generujeme strojový kód



Obrázek 1 - Postup kompilace [3]

Pro generování strojového kódu z mezikódu je potřeba nástroj *Clang,* což je překladač pro programovací jazyky C, C++, Objective-C a je součástí projektu LLVM. Ten umožňuje mít na vstupu kromě zdrojových kódů zmíněných jazyků právě i LLVM IR.

Kromě generování strojového kódu z mezikódu je v LLVM projektu k dispozici také JIT[2] kompilátor, který umožňuje kompilaci kódu při spuštění programu.

4 Cílová platforma

LLVM podporuje generování strojového kódu pro různé platformy. Seznam podporovaných platforem je možné vypsat příkazem:

```
llc --version
```

Tento příkaz zároveň vypíše aktuální konfiguraci, v našem případě byla pro veškeré testovaní použita tato konfigurace:

```
LLVM version 11.0.0
Default target: x86_64-pc-windows-msvc
Host CPU: skylake
```

Můžeme také říct, že „mezi platformou“ do které se bude náš jazyk překládat je LLVM IR, jehož instrukce je možné najít na [7].

[2] Just in time

5 Návrh jazyka

5.1 Zvolené konstrukce

Kromě minimálních požadovaných konstrukcí byly dále zvoleny tyto konstrukce.

Jednoduchá rozšíření (1 bod):

- for cyklus
- while cyklus
- do-while cyklus
- foreach cyklus pro pole
- else větev
- datový typ boolean
- datový typ real
- datový typ string
- násobné přiřazení
- ternární operátor
- funkce pro vstup a výstup

Složitější rozšíření (2 body):

- pole a práce s jeho prvky
- operátor pro porovnání řetězců
- parametry předávané hodnotou
- návratová hodnota podprogramu

Rozšíření vyžadující složitější instrukční sadu (3 body):

- dynamicky přiřazovaná paměť
- parametry předávané odkazem
- typová kontrola
- výstup pro reálnou platformu (.exe soubor pro Windows)

Další vlastnosti a požadavky na jazyk:

- nepovinné ukončení příkazů středníkem
- striktní typování
- při deklaraci proměnné je vždy nutné ji inicializovat
- break příkaz pro předčasné ukončení cyklu

5.2 Popis jednotlivých konstrukcí

Detailnější ukázky programů jsou ve složce *samples*, tento text je v obdobné podobě v git repositáři viz Uživatelská dokumentace.

5.2.1 Identifikátory

Identifikátor může být jakékoliv slovo obsahující písmena, číslice a podtržítko a musí začínat písmenem. Ve flexu je pro identifikátory použit tento regulární výraz: `[a-zA-Z][a-zA-Z0-9_]*`. Identifikátory jsou použity pro názvy proměnných a funkcí a jsou case sensitive.

5.2.2 Proměnné

Proměnné je možné deklarovat pomocí klíčových slov `var` a `val`.

- `var` se používá pro proměnné
- `val` se používá pro konstanty

Při deklaraci musí být proměnná vždy inicializována. Typ proměnné je automaticky dedukován dle inicializační hodnoty. Proměnná je viditelná pouze ve svém rozsahu (bloku) a může být překryta jinou proměnnou se stejným názvem v podbloku.

```
val a = 1 // constant
var b = 2 // reasigneable

func someFunction(Int x) { // x is var and can be reassigned
  val b = x // shadows first b
  x = 0
}
```

5.2.3 Datové typy

Dostupné datové typy jsou `Int, Double, Char, Bool, String`. Mlang je silně typovaný jazyk se statickou typovou kontrolou. Pro převod mezi typy jsou k dispozici vestavěnné funkce viz dále.

- `Int` je 64 bitový integer.

  ```
  val myInt = 1
  ```

- `Double` je 64 bitový float. Pro literály je možné použít tyto notace.

  ```
  val myDouble1 = 1.5       // 1.5
  val myDouble2 = .1        // 0.1
  val myDouble3 = 1.e2      // 100.0
  val myDouble4 = .1e-1     // 0.01
  ```

- `Char` je 8 bitový znak. Jsou dva způsoby jak vytvořit `Char` literál. Prvním je použití znaku `@` následovaného požadovaným znakem nebo použití dvou znaků `@` následovaných ascii hodnotou požadovaného znaku.

  ```
  val ch1 = @A
  val ch2 = @@65
  if (ch1 == ch2) { // always true }
  ```

- `Bool` je boolean, jsou dostupné literály `true` a `false`.

  ```
  val b1 = true
  val b2 = false
  ```

- `String` je interně reprezentován jako pole znaků a má všechny vlastnosti jako ostatní pole viz dále. `String` literál je uzvařen mezi `"` nebo `'` . Tyto znaky uvnitř `Stringu` musí být uvozeny znakem `\`.
  `String` může být vytvořen pomocí literálu nebo pomocí funkce `String(Int size)`. V tom případě je ale `String` vytvořen na haldě a musí být uvolněn pomocí klíčového slova `rm`.

  ```
  val str1 = "This is a \"string\"."
  ```

```
val str2 = String(10)
rm str2
```

Existuje speciální konstrukce pro spojování `Stringů`. Pro spojení dvou a více `Stringů` je potřeba je uzavřít mezi znaky `(.` a `.)` a oddělit je čárkou. Výsledný `String` je nutné nakonec uvolnit klíčovým slovem `rm`.

```
val res = (. str1, " and ", str2 .)
rm res
```

Jednotlivé znaky (typu `Char`) je možné získat pomocí `[index]` stejně jako u polí viz dále.

5.2.4 Pole

Pole jsou dostupná pro všechny výše zmíněnné datové typy (kromě `Char`, ale pro ten existuje `String`) a mohou být pouze jednorozměrná. Pole je vytvořeno voláním příslušné funkce s předáním požadované velikosti pole.

```
IntArray(10)
DoubleArray(10)
BoolArray(10)
String(10)
```

Pro přístup k položce na daném indexu je k dispozici výraz `[index]`. Velikost pole je interně uložena spolu s jednotlivými prvky pole a při přístupu na požadovaný index probíhá kontrola vůči jeho velikosti. Pro získání velikosti pole je dostupná funkce `sizeOf(array)`.

```
val arr = IntArray(10)
arr[0] = 1
arr[10] = 1 // runtime error index out of range
val i = arr[1]
```

Všechna pole jsou vytvářena na haldě a musí být řádně uvolněna pomocí klíčového slova `rm`.

```
rm arr
```

5.2.5 Funkce

Funkce jsou deklarovány pomocí klíčového slova `func` následovaného jménem funkce (identifikátor), parametry a typem návratové hodnoty. Tělo funkce je uvnitř složených závorek. Parametry jsou specifikovány svým typem a jménem a jsou to proměnné (jako `var`, lze do nich znovu přiřadit hodnotu). Návratová hodnota je pouze volitelná.

```
func myFunction(Int param1, DoubleArray param2): Bool {
  return true
}
```

Primitivní datové type (`Int, Double, Bool, Char`) jsou předávány hodnotou, pole (a `String`) jsou předávány odkazem.

```
func processArray(IntArray arr) {
  arr[0] = -1
}

val array = IntArray(10)
processArray(arr)
println(toString(arr[0])) // -1
```

Příkaz `return` je volitelný a potom je vrácen výsledek posledního výrazu.

```
func getInt(): Int { 1 }
```

5.2.6 Výrazy

- Přiřazení do proměnné je pomocí symbolu `=` . Výsledkem přiřazení je přiřazovaná hodnota, je tedy možné vícenásobné přiřazení.

  ```
  var a = b = c = 10
  ```

- Ternární operátor, který funguje jako v jazyce C.

  ```
  bool-expression ? then : else
  val a = b > 10 ? 1.5 : 5.1
  ```

- Boolean výrazy
  - Pro `Int, Double, Char a String`

    ```
    if (a == b) {...}
    if (a != b) {...}
    if (a > b) {...}
    if (a < b) {...}
    if (a >= b) {...}
    if (a <= b) {...}
    ```

  - Pro `Bool`

    ```
    if (true) {...}
    if (not true) {...}
    if (b1 and b2) {...}
    if (b1 or b2) {...}
    ```

- Binární operátory
  - Pro `Int, Double a Char`

    ```
    a + b
    a - b
    a * b
    a / b
    ```

  - Pro `Bool`

    ```
    a and b
    a or b
    ```

- Unární operátory
  - Pro `Int, Double a Char`

    ```
    i++   // return i and then increase i by one
    ++i   // increase i by one and return i
    -i
    ```

  - Pro `Bool`

    ```
    not b
    ```

5.2.7 Řídící konstrukce

- **If-else**, else větev je volitelná

```
if (bool-expression) {
      statements
} else {
      statements
}
```

- **While**

```
while (bool-expression) {
      statements
}
var i = 10

while (i > 0) {
      println(toString(i))
      i--
}
```

- **Do-While**

```
do {
      statements
} while (bool-expression)

do {
      i--
} while (i > 0)
```

- **For**

`For` cyklus funguje pouze s proměnnou typu `Int`, která je automaticky vytvořena a inicializována na počáteční hodnotu rozsahu (`Range`). Velikost kroku je volitelná a defaultně je `1`. Velikost kroku může být jak literál tak výraz.

```
for (var in range) { statements }
for (var in range step step-size) { statements }
```

Rozsah (`Range`) je specifikován začátkem, koncem (mohou to být jak literály tak výrazy) a jedním z klíčových slov `to` nebo `until`.

```
for (i in 0 to 10) {...}  // i in [1, 10], 10 is included
for (i in 0 until 10) {...}  // i in [0, 10), 10 is excluded

val start = 0
val end = 100
val s = 2
for (i in start to end step s) {
      println(toString(i)) // prints 0, 2, 4, .., 100
}
```

- **ForEach** je smyčka pro průchod jednotlivými položkami pole. Proměnná s hodnotou dané položky ja také automaticky vytvořena a typována na elementární typ pole.

```
for (var in array) { statements }

val array = IntArray(20)
for (i in array) {
      println(toString(i))
}
```

- **Break** je klíčové slovo, které může být použito pouze uvnitř cyklu (`for, foreach, while, do-while`) a slouží k předčasnému upuštění daného cyklu.

```
for (i in 0 to 10) {
      if (i > 5) {
            break
      }
}
```

5.2.8 Vstupní bod

V jazyce mlang může být jakýkoliv kód mimo funkce a je automaticky vykonán po startu programu. Pokud je ale potřeba, aby byla nějaká funkce volána automaticky, stačí ji pojmenovat `main`. Předávání argumentů funkci main bohužel v tuto chvíli nefunguje. Respektive není připravena konstrukce, která by dokázala zpracovat seznam řetězců (dvourozměrné `String` pole).

```
println("Outside 1") // called fist

func main() {
  // is called automatically
  println("Hello World") // called third
}

println("Outside 2") // called second
```

5.2.9 Import souborů

Importování dalších souboru se zdrojovými kódy jazyku mlang je možné pomocí klíčového slova `import` následovaného relativní cestou k danému souboru.

```
import other-file.mlang
import dir/another-file.mlang
```

5.2.10 Komentáře

Komentáře ve zdrojových kódech fungují stejně jako v jazyce C. Jednořádkové komentáře začínají `//` a všechny znaky do konce řádku jsou ignorovány. Víceřádkové komentáře začínají symboly `/*` a končí symboly `*/`. Všechny znaky mezi jsou ignorovány.

5.2.11 Terminátory

Gramatika jazyka používá středník jako terminátor výrazu, ale při lexikální analýze jsou automaticky vloženy (jako to dělá např. jazyk Go [5]), takže jsou středníky pouze volitelné.

5.2.12 Vestavěnné funkce

Existují vestavěnné funkce pro prostředkování (standardního) vstupu a výstupu a další pomocné funkce. Tyto funkce jsou implementovány v jazyce c a jejich binární podoba je v souboru *buildins.bc*, který je vygenerován při překladu a který musí být ve stejné složce jako soubor *mlang.exe*. Kromě níže uvedených „veřejných" funkcí obsahuje také interní funkce (začínající prefixem __mlang) a ty nelze volat přímo z kódu, ale jejich volání je do IR kódu vkládáno při jeho generování. Jedná se např. o pomocné funkce pro alokaci paměti a podobně. Naopak některé níže uvedené funkce v souboru *buildins.bc* nejsou a jejich volání je detekováno a obslouženo „manuálně" při generování kódu.

- **print** funguje jako *printf* v jazyce C. První parametr je typu `String` a udává formát řetězce, následuje proměnný počet argumentů pro daný formát viz [6].

  ```
  val name = "Martin"
  val age = 23
  print("Name: %s, age: %d", name, age)
  ```

- **println** je stejné jako předchozí `print`, ale přidá nakonec nový řádek (`\n`)
- **read** přečte jeden znak z klávesnice (`Char`)

  ```
  val ch = read()
  ```

- **readLine** přečte jeden řádek z klávesnice (`String`)

  ```
  val line = readLine()
  ```

- **sizeOf** vrací velikost pole nebo `Stringu`

  ```
  val arr = IntArray(10)
  val size = sizeOf(arr) // 10
  ```

- **len** vrací délku `Stringu` – počet znaků do znaku `\0` nebo konce `Stringu`

  ```
  val str = "Hello World"
  var l = len(str) // 11

  val str2 = String(20)
  l = len(str2) // 0
  val size = sizeOf(str2) // 20
  rm str2
  ```

- **cast funkce** pro převod mezi typy, pro všechny primitivný typy a `String` existují tyto funkce:

  ```
  toInt(...)
  toDouble(...)
  toBool(...)
  toChar(...)
  toString(...)
  ```

  Funkce `toString` v tomto případě alokuje String na zásobníku a proto není nutné volat `rm` pro uvolnění paměti.

6 Implementace

Veškeré zdrojové kódy se nachází v adresáři *mlang/src*. Ten obsahuje kromě *.cpp*/*.h* souborů ještě tyto soubory:

- **CMakeLists.txt** – obsahuje instrukce pro sestavení projektu pomocí nástroje Cmake
- **lexer.l** – je zdrojový kód pro *flex*, z něho se generuje soubor *lexer.cpp*
- **parser.y** – je zdrojový kód pro bison a obsahuje definici gramatiky jazyka, z něho se generuje soubor *parser.cpp*

6.1 Princip fungování

Zpracování vstupního souboru a komilace jsou rozděleny do několika kroků. Vstupním bodem programu je funkce *main* v souboru *main.cpp*.

6.1.1 Parsování vstupního souboru

Funkce main zpracuje vstupní argumenty a otevře vstupní soubor se zdrojovým kódem jazyku mlang a volá funkci *yyparse* ze souboru *parser.cpp*. Tato funkce poté vnitře čte jednotlivé tokeny pomocí funkce *yylex* ze souboru *lexer.cpp*.

6.1.2 Tvorba AST

Soubor *parser.y* obsahuje u jednotlivých pravidel gramatiky akce, které vytváří jednotlivé uzly AST. Kořenový uzel je uložen v proměnné *programBlock*. Zjednodušená verze gramatiky je v příloze A.

6.1.3 Generování IR

Pokud vše správně proběhne a vstupní soubor neobsahuje žádné syntaktické chyby proměnná *programBlock* obsahuje kořenový uzel AST. Ten je předán funkci *generateCode* z třídy *mlang::CodeGenContext*.

V této funkci se nejprve generuje kód pro pomocnou inicializační funkci (*__mlang_init_fun*), která obsahuje všechen kód zapsaný mimo funkce. Poté se „zaregistrují“ vestavěnné funkce. A následně se začíná generovat kód pro jednotlivé uzly AST.

6.1.4 Detekce chyb

Pokud se v průběhu generování kódu objeví nějaká chyba (např. nekompatibilita typů, neexistující funkce apod.), vypíše se popis chyby společně s pozicí chyby ve zdrojovém souboru a navýší se proměnná *errors*.

6.1.5 Verifikace

Poté co je vygenerován IR je spuštěna jeho verifikace pomocí funkce *llvm::verifyModule*, která může odhalit další potencionální chyby, která se během generování odhalit nepodařilo.

6.1.6 Optimalizace

Po verifikaci se ještě spustí optimalizace IR (pouze pokud není nastaven přepínač *debug* viz Uživatelská dokumentace) pomocí třídy *llvm::legacy::FunctionPassManager*.

6.1.7 Spuštění programu

Pokud je při spuštění nastaven přepínač *run* (viz Uživatelská dokumentace) je program ihned spuštěn pomocí třídy *llvm::ExecutionEngine* s parametrem *llvm::EngineKind::JIT*.

6.1.8 Tvorba exe souboru

V případě, že přepínač *run* není nastaven, uloží se vygenerovaný (a optimalizovaný) IR kód do souboru. Následně se (automaticky) vytvoří spustitelný exe soubor pomocí nástroje *Clang* a příkazu:

```
clang –x ir –o fileName.exe fileName.mlang.ir path_to_mlang/buildins.bc
```

6.2 Příklad generování

Knihovna LLVM obsahuje pomocné třídy a funkce pro generování jednotlivých instrukcí, níže jsou uvedené příklady jak vygenerovat některé instrukce. Většině těchto pomocných tříd se předávají některé z těchto typů argumentů:

- **context** typu *llvm:LLVMContext*, který vlastní a spravuje globální data jádra LLVM
- **block** typu *llvm::BasicBlock,* který představuje blok instrukcí, které jsou sekvenčně vykonávány
- **value** typu *llvm::Value*, jedná se o základní třídu všech hodnot vypočítaných programem a lze ji použít jako operand pro jiné instrukce. Jedná se o rodičovskou třídu dalších důležitých tříd jako je např. *Instruction*, *Block*, *Function,…*

6.2.1 Skok

Skok (v tomto případě nepodmíněný) provedeme vytvořením nového bloku:

```
auto bb = llvm::BasicBlock::Create(context, "after");
```

A vlastní skok na tento blok (instrukce `br`) provedeme pomocí:

```
llvm::BranchInst::Create(bb, context);
```

6.2.2 Konstanta

Konstantu (např. 64 bitový integer) vytvoříme příkazem (poslední parametr udává, zda se jedná o signed nebo unsigned konstantu):

```
lvm::ConstantInt::get(llvm::Type::getInt64Ty(context), hodnota, true);
```

6.2.3 Binární operátor

Binární operátor, např. sečtení dvou integerů provedeme pomocí:

```
llvm::BinaryOperator::Create(llvm::Instruction::Add, value1, value2, "mathtmp",
context);
```

Výsledkem všech těchto volání je hodnota typu *llvm::Value*, která lze použít v dalších instrukcích jako jejich parametr.

6.3 Uživatelská dokumentace

6.4 Popis archivu

Odevzdaný archiv obsahuje tyto soubory:

- **doc**
  - **doc.pdf** – tato dokumentace
- **mlang**
  - **src** – zdrojové kódy
  - **samples** – ukázkové vstupní soubory
  - **build**
    - **build.bat** – skript pro překlad
- **release** – obsahuje binární podobu programu *mlang.exe*

6.5 Odkaz na Git

Projekt je uložen v git repozitáři na serveru *github.com*, kde se navíc také nachází jednodušší obdoba této dokumentace. Odkaz na repositář: https://github.com/MFori/mlang

6.6 Požadavky

Pro sestavení projektu je nutné mít nainstalované tyto nástroje:

- LLVM verze 11.0.0
- Clang (součástí LLVM 11.0.0)
- Flex verze 2.6.4
- Bison verze 3.7.1
- CMake verze 3.17
- Visual Studio (16 2019)

Dále musí být nastavené proměnné prostředí:

- Path musí obsahovat cestu k souborům *clang.exe*, *flex.exe*, *bison.exe* a *cmake.exe*
- LLVM_DIR a LLVM_ROOT musí obsahovat cestu k souboru *LLVMConfig.cmake*

Postup pro instalaci LLVM a Clang je k dispozici v repositáři:
https://github.com/MFori/mlang/blob/main/llvm-setup.md

6.7 Sestavení

Pro sestavení je k dispozici soubor *build/build.bat*, který vykoná tyto příkazy:

```
cmake .. -G „Visual Studio 16 2019“
cmake --build . --config Release
```

Přepínač -G lze nahradit jiným generátorem, ale testováno bylo pouze s *Visual Studio 16 2019*. Seznam dostupných generátorů lze zobrazit příkazem:

```
cmake -help
```

Druhý příkaz ze souboru *build.bat* lze nahradit otevřením souboru *mlang.sln* (který je vygenerován prvním příkazem) a zkompilovat projekt ve Visual Studiu.

Výsledkem je soubor *mlang.exe* (a další soubory). Výstupní adresář je závislý na předchozím postupu může to být např. *build/src/Release* nebo *cmake-build-release/src*.

Poznámka: Někdy se stane, že se při použití *build.bat* překlad „sekne“, potom stačí dát enter a pokračuje dál, nepodařilo se mi zjistit, čim to je.

6.8 Použití

Nyní stačí vytvořit soubor se zdrojovým kódem jazyku mlang (např. *samples/hello_world.mlang*) a spustit soubor *mlang.exe*, který má tyto parametry:

- **-h (--help):** zobrazí nápovědu
- **-d (--debug):** vypne optimalizaci
- **-r (--run):** rovnou spustí program (tzv. Just In Time kompilace)

Pokud je přepínač *run* vypnutý je výstupem soubor s LLVM IR kódem (*hello_world.mlang.ir*) a spustitelný soubor (*hello_world.exe*).

7 Závěr

Podařilo se navrhnout gramatiku jazyka a vytvořit překladač pomocí nástrojů flex, bison a LLVM. Pro otestování bylo vytvořeno několik scénářů, které demonstrují, že si kompilátor dokáže poradit s jednoduššími, ale i se složitějšími konstrukcemi jako je rekurzivní volání.

Možným vylepšením by bylo přidat podporu pro předávání parametrů vstupní funkci (main), ale k tomu by bylo potřeba přidat také podporu vícenásobných polí.

Reference

[1] LLVM Project. Llvm.org [online]. [cit. 2020-12-15]. Dostupné z: https://llvm.org/docs/index.html

[2] LLVM Tutorial. Llvm.org [online]. [cit. 2020-12-15]. Dostupné z: https://llvm.org/docs/tutorial/

[3] Creating a programming language with C++ and LLVM. Https://guiferviz.com/ [online]. [cit. 2020-12-15]. Dostupné z: https://guiferviz.com/uranium/00-introduction/

[4] Flex_lexical_analyser. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-200 [cit. 2020-12-15]. Dostupné z: https://cs.wikipedia.org/wiki/Flex_lexical_analyser

[5] Golang.org [online]. [cit. 2020-12-15]. Dostupné z: https://golang.org/ref/spec#Semicolons

[6] Printf. Cplusplus.com [online]. [cit. 2020-12-15]. Dostupné z: http://www.cplusplus.com/reference/cstdio/printf/

[7] Instruction reference. Llvm.org [online]. [cit. 2020-12-16]. Dostupné z: https://llvm.org/docs/LangRef.html#instruction-reference

A Gramatika jazyka

program : %empty | stmts

stmts : stmt | stmts stmt

stmt : expr ';' | ';' | var_decl ';' | func_decl | conditional | return ';' | break ';' | free ';' | while | for

lstmt : expr | var_decl | return | break | free

block : '{' stmts '}' | '{' stmts lstmt '}' | '{' '}'

primary_expr : ident | ident '(' call_args ')' | '(' expr ')' | literals | '(.' call_args '.)'

postfix_expr : primary_expr | postfix_expr '[' expr ']' | postfix_expr TINC | postfix_expr TDEC

unary_expr : postfix_expr | TINC unary_expr | TDEC unary_expr | TNOT unary_expr
| TMINUS unary_expr | TPLUS unary_expr

binop_expr : unary_expr | binop_expr TPLUS unary_expr | binop_expr TMINUS unary_expr
| binop_expr TMUL unary_expr | binop_expr TDIV unary_expr

compare_expr : binop_expr | compare_expr TCEQ binop_expr | compare_expr TCNE binop_expr
| compare_expr TCLT binop_expr | compare_expr TCLE binop_expr
| compare_expr TCGT binop_expr| compare_expr TCGE binop_expr

and_expr : compare_expr | and_expr TAND compare_expr

or_expr : and_expr | or_expr TOR and_expr

ternary_expr : or_expr | or_expr '?' expr ':' ternary_expr

assignment_expr : ternary_expr | unary_expr '=' assignment_expr

expr : assignment_expr

call_args : %empty | expr | call_args ',' expr

var_decl : TVAR ident '=' expr | TVAL ident '=' expr

func_decl : TFUNDEF ident '(' func_decl_args ')' ':' ident block
| TFUNDEF ident '(' func_decl_args ')' block

func_decl_args : %empty | ident ident | func_decl_args ',' ident ident

return : TRETURN | TRETURN expr

break : TBREAK

free : TFREE expr

ident : TIDENTIFIER

literals : TINTEGER | TDOUBLE | TSTR | TBOOL | TCHAR

conditional : TIF '(' expr ')' block TELSE block | TIF '(' expr ')' block

while : TWHILE '(' expr ')' block | TDO block TWHILE '(' expr ')'

for : TFOR '(' ident TIN range ')' block | TFOR '(' ident TIN range TSTEP expr ')' block
| TFOR '(' ident TIN expr ')' block

range : expr TUNTIL expr | expr TTO expr