Szűcs Dániel – BSY9BQ

Szeged, 2023.03.31.

1.	1 - 40 óra
2022. 07. 04. – 2022. 07. 08.	

A szakmai gyakorlatomat az SZTE Szoftverfejlesztés tanszéken végeztem, ahol C++ fejlesztői pozícióban munkálkodtam. Felvételem után az első 40 órában betanuláson vettem részt. A betanulás első részében általános információkkal ismerkedtem meg, mint például, hogyan tudom elérni a belső szervereket, hogyan tudom rögzíteni a munkaóráimat, hol érem el a munkahelyi chatet stb. Ezek után a betanulásra kapott számítógépre kellett Ubuntu-t telepítsek. A betanulás első témája a Linux alapok volt. Itt a GNU szoftvercsomag programjairól tanultam részletesen a BASH-ról, majd a GNU coreutils többi programjáról például: cd, chmod, grep, sudo, make, xargs. Ezen felül tanultam még a Linux kernelről, hogyan tudok magamnak kernelt fordítani, és hogyan lehet kernel modulokat betölteni. A következő részben a különböző függvény könyvtárakról tanultam, mint például a glibc és a libstdc++. Ezen felül tanultam még a virtualizációról is és telepítettem VirtualBox-ba egy Arch Linux disztribúciót próbaként. A második téma betanulásom során a Git verziókezelő rendszer volt. Itt megtanultam hogyan kell bare (remote) repositoryt inicializálni, hogyan kell local repositoryt inicializálni, hogyan tudom a nevemet és e-mail címemet beállítani, hogyan érem a logot stb. Majd a Git komplexebb parancsairól beszéltünk, itt megtanultam hogyan működik a branching, mi az a rebase, merge, cherry-pick, hogyan tudom a változtatásaimat committolni majd feltenni a remote-ba, hogyan tudok conflict-ot feloldani. A harmadik téma a betanulás alatt a GNU toolchain volt. Itt tanultam a gcc fordítóprogramról és különböző kapcsolóiról. Megnéztem, hogy a fordító hogyan rendezi el a programomat memóriában, a különböző változók és a kód milyen bináris szekciókba kerülnek. Megtanultam mi az Address Space Layout Randomization (ASLR). Megtanultam, hogy a toolcahin milyen komponensekből áll, és hogy a komponensek között milyen kapcsolat áll. Ezek után a linker komponenssel foglalkoztam többet, ahol megnéztem hogyan tudok linker scriptet írni, hogyan tudom külső könyvtárak elérési útját megadni neki stb. A betanulás utolsó témája a debugging volt. Itt tanultam külöböző debuggolásra alkalmas programokról, mint például a gdb, cgdb, lldb. Megtanultam hogyan kell breakpointokat és watchpointokat használni, valamint hogyan lehet kivételeket elkapni debuggolás közben.

2.	41 - 80 óra
2022. 07. 11. – 2022. 07. 15.	

A betanulást követően a munkám egy másik irodában folytattam, ahol megismerkedtem a projekten dolgozó kollégákkal. Első feladatom a számítógépem installálása volt, melyet a hálózat beállítása követett, hogy el tudjam érni VPN-en keresztül a belső hálózatot. Ezután létre kellett hoznom egy új felhasználót, amivel hozzáférek a belső hálózaton elérhető dolgokhoz. Az első programozási feladatom a projekten az volt, hogy statikus analízis segítségével kód hibákat kellett kijavítsak. Ehhez megismertem a Clang-Tidy nevű eszközt, amely a kiemeli a kódban található potenciális problémákat. Mivel a projekt teljes szkennelése körül belül 15 percet vesz igénybe így inkább iteratív módon próbáltam dolgozni, ez viszont csak részben lehetséges a komponensek összefüggése miatt. Ezt a feladatot azért kaptam, hogy a projekt teljes egészét át tudjam tekinteni. A projekten fejlesztett program egy fordítóprogram, ami öt fő komponensből áll, név szerint: Lexer, Parser, Checker, Compiler, Emitter. A következő feladatom során a GitLabon feljegyzett issuekkal kellett foglalkoznom a Parser és a Lexer komponensben. A legtöbb eseteben a hiba az volt, hogy a program rossz sor információt adott vissza, ha valami hibát talált a forráskódban. Ezeket gyorsan tudtam is javítani. Minden issue javítása után szólnom kellett a mentoromnak, aki átnézte a változtatásaimat, majd elmondta a meglátásait. Miután mindent rendben talált feltehettem a kódomat a közös branchre, majd lezártam az issue-t. A következő feladatom az volt, hogy a valósítsam meg a feltételes vezérlési szerkezet lefordítását. Ehhez meg kellett ismerjem a Compiler komponenst. A feltételes vezérlési szerkezet fordítása a következőképpen történik: Először megnézzük, hogy az *if* feltétele fordítási időben kiértékelhető-e, mert ha igen akkor magát a vezérlési szerkezetet le sem kell fordítani ezzel runtime időt és teljesítményt nyerve. Ezek után történik a tényleges fordítás. Először is szükség van egy *label-re* ahová akkor ugrunk, ha az *if* feltétele hamis. Ezután lefordítjuk az *if* feltételét melynek allokálnunk kell egy regisztert, aminek értékét a nullával fogjuk összehasonlítani, majd a végén le kell még fordítani konzekvens utasítást is. Ha van *else* akkor frissíteni kell a *label-t,* hogy a megfelelő helyre történjen a vezérlés átadása. Ha *else if* követi a konzekvens utasítást akkor rekurzívan hívjuk az *if-et* fordító függvényt. A Compilerben vissza kell állítani a következő *label-re* mutató pointert és ezzel gyakorlatilag kész a feltételes vezérlési szerkezet lefordítása.

3.	81 - 120 óra
2022. 07. 18. – 2022. 07. 22.	

A következő feladatom különböző logikai bináris operátorok lefordítása volt. Ezek az operátorok a következők: ==, !=, >=, <=. Ezeknek az operátoroknak a lefordítása páronként nagyon hasonlóan történik, ezért a kódomat template-ekkel oldottam meg, hogy csökkentsem a kód duplikációt. Mind a négy operátor esetében az első lépés a feltételes vezérlési szerkezethez hasonlóan azzal a lépéssel kezdődik, hogy megnézzük, hogy fordítási idejű konstansok-e, ha igen akkor az előre kiszámolt értéket töltjük be az akkumulátorba. Mivel ezek az operátorok mind logikai értéket adnak ezért ez igaz érték esetén 1, hamis érték esetén 0. A template függvények template paraméterben egy *flag-et* várnak, ami fordítási időben eldönti, hogy melyik operátor kódját szintetizálja. Ezt a munkámat is átnézete a mentorom majd feltöltöttem a közös branchre. Ettől a ponttól kezdve szabadon dolgozhattam, nem kellett a kódomat bemutatni a mentoromnak, hanem a hagyományos code review folyamaton esett át. A következő feladatom a Checker komponenst érintette, ami a projekt legbonyolultabb komponense. A Checker komponens végzi a szemantikai kontextus függő analízist a kódon. Mivel a nyelv, amit a fordítóprogramunk fordít statikusan típusos, és objektum orientált, így nagyon sok különböző szemantikai analízisen esik át. Az én feladatom a *class-okat* érintette, azon belül is az *abstract* függvényekre kellett statikus analízist implementálnom. Az analízis a következő szabályok mentén működik: *abstract* függvénye csak *abstract* osztálynak lehet, *abstract* osztály nem példányosítható, nem *abstract* osztály, ha *abstract* ősből származik akkor köteles minden *abstract* függvényének implementációt adni. Ezek az analízisek relatív könnyen implementálhatók voltak. A következő feladat, amelyet már magamnak választhattam az Emitter komponenst érintette. A fordítóprogram a futása végén nem platform specifikus assembly kódot emittál, hanem bytecode-ot. Ebben a bytecode-ban elhelyezhetők különböző féle metainformációk amelyeket a nyelvhez fejlesztett runtime és debugger fel tudnak használni. A feladat lényege az volt, hogy minden egyes generikus függvényhez, vagy osztályhoz ki kellett emittálni az bytecode-ba, hogy mennyi típus paramétere van, és azoknak van-e egymás között függőségük vagy felső korlátjuk. Ehhez tulajdonképpen minden információ meg van az adott AST node-on, így az Emitter gyakorlatilag csak egy szerializációs feladatot lát el. A szerializáció során, minden adattagot *string-é* alakít.

4.	121 - 160 óra
2022. 07. 25. – 2022. 07. 29.	

A projekten ezen a ponton nagyon sok munka volt. Viszont a legfontosabb az volt, hogy a belső hálózaton lévő GitLab szerverről át kellett költöztetnünk a projektet egy másik git hostingra, ahol már, mint open-source projektet fejlesztettük tovább. A projekt migrációja során sok gond felmerült. Voltak olyan elnevezések, licenszek, szimbólumok, amelyek továbbra is a belső hálózaton lévő dolgokra hivatkoztak, ezeket meg kellett változtassam. Voltak helyek a kódban, ahol egyszerűen csak kicseréltem az elnevezéseket, voltak olyanok is viszont, ahol ez nem volt elegendő így komolyabb kód refaktorálást kellett végezzek. Több a belső hálózatról származó dependenciát is ki kellett szervezzek a kódból, ezeket helyenként leimplementáltam kézzel, helyenként pedig más nyíltforráskódú dependenciára cseréltem. A legnagyobb falat viszont az volt, hogy a projekt egy része még a felvételem előttről már kint volt open-source-ban viszont ezt a projektet egy másik csapat fejlesztette egy darabig, majd a fejlesztésével felhagytak. Az volt a feladatom, hogy a mi fejlesztésünket, ami körül belül négy-öt hónappal az open-source fork előtt járt, valahogy rá kellett rebaseljem az open-source változatra. Ez nagyobb nehézségeket hozott magával, mint bármilyen Git művelet, amit eddig végeztem. Kezdésnek mivel az open-source változat fejlesztése abba maradt így annak csak részeit kellett megtartanom. Ezt úgy végeztem, hogy az ott lévő commitok közül azokat választottam ki, amelyek stabil funkcionalitásokat hoztak be, valamint megfeleltek az addigra már öt hónappal frissebb szabványnak. Ezek után megpróbáltam naivan rebaselni a még belső GitLab-os repositoryt az open-source repositoryra. Sajnos ez nem sikerült mert a kettő repository között a rebaselést csak egy patch fájl generálásával tudtam végezni, viszont megváltoztak fájl útvonalak ezért ez a patch nem volt alkalmazható egyszerűen. Ezen a ponton segítséget kértem a kollégáimtól, hogy hogyan lehetne ezt a különbséget a két repository között megoldani anélkül, hogy a belső repository-ból több különböző patch fájlt generáljak. A megoldás végül egy komplex Git parancs volt. A Git apply parancsának megadható két olyan kapcsoló, hogy a patch fájlban látható elérési utat bontsa fel, könyvtár delimiterenként és vegyen le belőle valamennyi könyvtárat majd fűzzön hozzá valamennyi könyvtár nevet. Ezzel a paranccsal végül sikerült megoldani a problémát így következhetett a Jenkins continuous integration (CI) beüzemelése, majd az átállás branch modellről, fork modellre.

5.	161 - 200 óra
2022. 08. 01. – 2022. 08. 05.	

A következő feladatom a projekt open-source-ba kerülése után a *long* literálok parsolásának a megjavítása volt, amely bináris, oktális, és hexadecimális literálok esetén hibás volt. A probléma forrása egy integer túlcsordulás volt, amit egy rosszul felparaméterezett template függvény okozott. A függvény lényege, hogy megpróbál végig menni egy bináris, oktális, vagy hexadecimális literálon majd visszaadja az annak megfelelő decimális literált. Ez a függvény eredetileg csak a JavaScript nyelvi standardra volt felkészítve, így nem is várt mást, mint *double-t* mivel a JavaScriptben nincs integer típus. Ennek ellenére a függvény jól működött 32 bites integer esetén is viszont sosem volt meghívva úgy, hogy az eredménye 64 bites literált adjon vissza. Az általunk fejlesztett szkriptnyelv esetében sajnos nincs ’l’ vagy ’L’ literál szuffixum így egy *int* literál a *long* literáltól csak méretében különböztethető meg. A megoldásom erre a problémára a következő, elmentem a Lexer pozícióját mielőtt elkezdeném beparsolni a literált, ezt követően megpróbálom 32 bites integer literálként beparsolni. Ha ez a művelet nem sikerül akkor visszatekerem a Lexert az elmentett pozíciójára, majd megpróbálom beparsolni 64 bites integer literálként. Ha ez sem sikerül akkor az azt jelenti, hogy a literál nem helyes, például illegális karaktereket tartalmaz, vagy kettő vagy több helyiértékeket elválasztó alul vonás szerepel a benne. A következő feladatom a JavaScript nyelv egyik operátorának parsolása volt. Ez az operátor az optional chain (?.) operátor volt. Ez az operátor arra jó, hogy ha a baloldali operandusa *null* vagy *undefined* akkor a jobb oldali operandusát nem értékeli ki helyette, *undefiend-t* ad vissza. Az operátor parsolása már meg volt valósítva, viszont rosszul működött abban az esetben, ha a jobb oldali operandusa egy privát osztály mező volt. A privát osztály mező a ’#’ szimbólummal kezdődik amit egy identifier követ. A probléma abból eredt, hogy az optional chain parsolásának a kezdetén le lett mentve, hogy a Lexer feje milyen tokenre mutat, ezt követően meg volt vizsgálva, hogy az aktuális token ’#’-e, ha igen akkor a Parser levalidálta, hogy ebben a környezetben használható-e privát osztály mező. Ez a művelet azonban eltolta a Lexer fejét egy tokennel, így a Lexer lementett pozíciója a jelenlegi pozíció mögött volt egyel. Ebben a kontextusban az optional chain parse függvénye nem tudta kezelni a következő tokent, mert a lementett pozíció még mindig a ’#’-ra mutatott aminek a validálása már megtörtént. A megoldás, hogy a token lementését sort a validálás után kellett megtenni.

6.	201 - 240 óra
2022. 08. 08. – 2022. 08. 12.	

A következő feladatom, az egyik legnagyobb feladat volt, amit a projekten végeztem. A probléma komplexitása miatt több lehetséges megoldást is kidolgoztam, végül találtam egyet, amely zéró kompromisszumos volt, viszont implementációja a legbonyolultabbnak bizonyult. A probléma a Compiler komponenst érintette, ahol, ha 16-nál több lokális változónk volt és a 17. változóval próbáltunk függvényt hívni akkor elromlott a regiszter spillelés. A regiszter spillelésre azért van szükség mert a bytecode a hívásoknál regiszterekben adja át a paramétereket, viszont csak 16 regiszter allokálására képes, így, ha egy argumentuma a 17. regiszterben volt akkor azt a regisztert át kell mozgatni a 1-16 regiszterek valamelyikébe, viszont azok a regiszterek is tartalmazhatnak értékeket, így a kiválasztott regiszter értékét is (ha van) ki kell spillelni egy 17 vagy afölötti regiszterbe. Mivel a bytecode típusos így a Compilernek nyilván kell tartania, hogy melyik regiszterben milyen típusú változó van. A probléma pontosan abból adódott, hogy a Compiler a regiszter reprezentációjában tárolta, a típust is. Ez azért volt problémás mert vannak olyan függvény hívások, ahol például nulla paramétert adunk át, viszont a bytecode ezt nem támogatja, csak a kettő, négy, és N paraméteres hívásokra van beépített utasítás. Ez azt jelenti, hogy a többi esetben kénytelen a Compiler invalid regisztereket átadni a hívásoknak, azonban erre a regiszter spiller nem volt felkészítve, és ezeknek az invalid regisztereknek is követte a típusát. Az első naiv megoldásom az volt, hogy invalid regiszterek helyett allokáltam tényleges valid, azonban üres regisztereket. Ez nyilván valóan futás idejű lassuláshoz vezet, viszont ez csak optimalizálatlan kódra igaz. A Compiler által előállított bytecode-ot, ha megadtuk a toolchain-nek akkor optimalizálni is lehet, ezt a Byte Code Optimizer (BCO) végzi. Mivel az optimizer átlátja a regiszterek közötti függőségeket így az opt-level 1-n és 2-n lefordított programban az üres regiszterek allokálása nem történik meg csak opt-level 0-s esetben. Ezt a megoldásomat viszont nem tartottam kellően optimálisnak, mert DEBUG buildben az üres regiszterek zavaróak lehetnek a debuggolást végző felhasználó számára, vagy magát a debuggert kellene felkészíteni arra, hogy az üres regisztereket kezelje. Ez egy implementációs részlet miatti összefonódást eredményezne a compiler és a debugger között, amit mindeképpen el szerettem volna kerülni. A tényleges jó megoldást csak a következő héten találtam meg.

7.	241 - 280 óra
2022. 08. 22. – 2022. 08. 26.	

Az optimális megoldás a következő volt. A regiszterek helyett egy HashMap-ben tároltam el minden regiszter típusát. Mivel a függvények fordítása párhuzamosan történik így minden szálnak saját HashMap-e van. Ez megoldja a regiszterek típusának pontos követését, viszont a hívásoknál még mindig kénytelen a Compiler invalid regisztereket átadni, ha az argumentumok száma nem egyezik meg az utasítás által várt argumentumok számával. Erre a megoldást az jelentette, hogy refaktoráltam a regiszter spiller-t, hogy mielőtt egy hívás utasítást kiemittálna, és elkezdené spillelni a regisztereket, nézze meg, hogy az átadott argumentumok közül ténylegesen mennyi, ami nem invalid. Ezen ismeretekkel a regiszter spiller, el tudja látni feladatát zökkenő mentesen. A következő feladatomban egy egyszerűbb problémával foglalkoztam. Mivel a compiler toolchain lefordítható a legtöbb platformra, köztük Androidra is, így a fejlesztés során sok különböző dologra oda kell figyelni, és sok feltételes fordítást kell végeznünk makrókkal. Az egyik ilyen rész a kódban a Parser komponenshez kötött fájlrendszer bejárás, ahol a nyelv standard függvénykönyvtárát keressük. Ez a függvény minden platform esetén a C++17 óta elérhető *std::filesystem* implementációval megoldott, kivéve Androidos platformon, mert az Android Native Development Kit 21 nem implementálja a C++17 szabvány ezen függvénykönyvtárát. A megoldás eddig egy C-ből származó könyvtár használata volt. Én ezt a kódot refaktoráltam úgy, hogy minél kevesebb makrót kelljen használni benne, illetve megszűntettem a felesleges runtime hívásokat. Ezt követően egy érdekes feladatot kaptam. Az általunk fejlesztett szkriptnyelvben megengedettek a globális függvények és globális változók, viszont ez csak a látszat ugyanis ezek a függvények és változók egy láthatatlan *Global* nevű osztály statikus tagjai. Erre azért van szükség mert a bytecode nem támogat szabadon álló függvényeket és változókat, ezeknek mindig class-hoz kötve kell szerepelniük. Ahhoz viszont, hogy a globális változók inicializálódni tudjanak, generálnunk kell az *Global* class-hoz egy *class intializer-t*. Minden class esetében a *class initializer* felelős azért, hogy minden statikus változó értéket kapjon. Ennek a generálása viszont elmaradt az *Global* esetében. Arra viszont oda kellett figyeljek, hogy az általunk fejlesztett szkriptnyelv standard függvénykönyvtárból érkező szimbólumoknak ne generáljak extra *class initializer-t.* A feladat ettől a ponttól kezdve triviális volt, pár sor refaktorálást igényelt.

8.	281 - 320 óra
2022. 08. 29. – 2022. 09. 02.	

A következő feladatom a Checker komponenst érintette. A feladat az azonosítók feloldásának újra dolgozása volt. Mivel a toolchain fejlesztése párhuzamosan zajlik az általunk fejlesztett szkriptnyelv szabványának írásával, így előfordul, hogy egy a toolchainben már leimplementált működésen változtatni kell. A változtatás jelenleg az volt, hogy a sima azonosítókat mindig a globális szkópban kell feloldani. Ez abban tért el az eddigi működéstől, hogy eddig minden azonosító úgy volt feloldva, hogy mindig a legbelsőbb szkóptól indulva végig iterálva a szkóp láncon megkerestük az első nevet, amely egyezett az azonosítóval. Class-ok esetén, ha egy példányhoz tartozó adattagot akarunk elérni akkor azt a *this* kulcsszóval tehetjük meg, ha egy statikus adattagot akarunk elérni akkor pedig az osztály nevén keresztül érhetjük el. Az eddigi logikát úgy módosítottam, hogy letiltottam a sima azonosítókra az osztályon belüli keresést. Ezt úgy értem el, hogy először, ha függvényben vagyok akkor a függvény lokális változói majd argumentumai között keresek, ha ez nem járt sikerrel akkor feliterálok a szkóp láncon az első nem class szkópig majd innen kezdek egy iteratív keresést. Ez önmagában már helyes működést biztosít viszont, hogy hasznos hibaüzeneteket tudjon adni a fordítóprogram, így, ha nem sikerült ez a feloldás akkor megpróbálom az osztályban is feloldani a keresett azonosítót. Ha megtalálom az osztályban akkor egy segítőkész hibaüzenetet tudok adni a felhasználónak, hogy az általa keresett szimbólum létezik, csak hibásan próbál rá hivatkozni. Mivel a feloldás megváltoztatása és az új hibaüzenetek rengeteg konformancia és regresszió teszteket elrontottak, amelyek még nem voltak frissítve a szabvány módosítás óta, így a feladat maradék részében ezeket a teszteket javítottam ki, valamint adtam hozzá új pozitív és negatív teszteket is a jobb lefedettség érdekében. Az egyik ilyen teszt írása közben sikerült is a megvalósításomban találni egy hibát. A hiba forrása az volt, hogy a Checker komponens tényleg helyesen validálta le az azonosító feloldását, viszont a Compiler komponens is végzett szkóp lánc bejárást, mivel egy értékadás fordítása közben tudnia kell, hogy a bal oldalon álló azonosító mire mutat. Ez lehet egy lokális változó, vagy egy osztály mező is, ennek tekintetében más-más bytecode-ot kell a Compiler-nek előállítania. Mivel a két komponensnek ugyan azt az algoritmust kell használniuk így ezt a működést kiszerveztem magába a *scope* implementációjába.