Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Szűcs Dániel

2023

Szegedi Tudományegyetem

Informatikai Intézet

Es2panda fordítóprogram fejlesztések

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | Szűcs Dániel |  | Dr. Kiss Ákos |  |
|  | informatika szakos hallgató |  | egyetemi docens |  |

Szeged

2023

Feladatkiírás

A témavezető által megfogalmazott feladatkiírás. Önálló oldalon szerepel.

Tartalmi összefoglaló

* A téma megnevezése:

Es2panda fordítóprogram fejlesztése, az ETS nyelvi konformancia növelés érdekében.

* A megadott feladat megfogalmazása:

A cél, hogy az Es2panda fordítóprogram funkcionalitását kibővítsem a különböző komponensei fejlesztése által. A Checker komponensnek képesnek kell lennie különböző, az objektum orientált programozási paradigma által leírt szemantikai elemzésekre, továbbá az ETS nyelv specifikus szkóp feloldási szabályainak kezelésére. A Compiler komponensnek képesnek kell lennie a feltételes vezérlési szerkezet és a különböző bináris operátorok bytecode-ra fordítására, valamint függvény szkópokban a paraméterek és lokális változók regiszter allokálására és spillelésére.

* A megoldási mód:

Először is, a Compiler komponens egyszerűbb részein, a vezérlési szerkezeteken és az operátorokon, majd a Checker komponens OOP mintáinak szemantikai elemzésén dolgoztam, így átfogó ismeretet szereztem a komponensek felépítéséről. Ezek után a két komponenst párhuzamosan fejlesztettem, az új nyelvi elemek implementációs sajátosságainak megfelelően.

* Alkalmazott eszközök, módszerek:

Az Es2panda fejlesztése Ubuntu 22.04 operációs rendszeren folyik, a C++ nyelv 17-es szabványában. A fejlesztéshez Visual Studio Code fejlesztői környezetet használok. A fordításhoz az LLVM toolchain 14-es verzióját használom, debuggoláshoz a cgdb eszközt választottam.

* Elért eredmények:

Implementáltam egy robosztus rendszert a regiszter spillelésre, valamint a regiszterek statikus típusainak követésére, továbbá implementáltam a szkóp feloldást, amely felhasználóbarát fordítási hibaüzeneteket ad a fordítóprogram felhasználójának.

* Kulcsszavak:

fordítóprogram, szemantikai analízis, regiszter spillelés

Tartalomjegyzék

[Feladatkiírás 2](#_Toc133846891)

[Tartalmi összefoglaló 3](#_Toc133846892)

[Tartalomjegyzék 4](#_Toc133846893)

[1. Bevezetés 5](#_Toc133846894)

[2. Mi az az Es2panda? 6](#_Toc133846895)

[2.1. Mi az a fordítóprogram frontend? 6](#_Toc133846896)

[2.2. Mi az az Es2panda? 6](#_Toc133846897)

[2.3. Hogyan fejlesztjük az Es2pandát? 6](#_Toc133846898)

[3. Az Es2panda struktúrája 7](#_Toc133846899)

[3.1. A fizikai struktúra 7](#_Toc133846900)

[3.2. A logikai struktúra 8](#_Toc133846901)

[3.2.1. A Lexical Analyser (Lexer) 8](#_Toc133846902)

[3.2.2. A Parser 9](#_Toc133846903)

[3.2.3. A Binder 9](#_Toc133846904)

[3.2.4. A Semantical Checker 10](#_Toc133846905)

[3.2.5. A Code Generator 10](#_Toc133846906)

[3.2.6. Az Emitter 10](#_Toc133846907)

[3.2.7. Az Absztrakt szintaxis fa (AST) 11](#_Toc133846908)

[4. A Parser 12](#_Toc133846909)

[4.1. A Parser belépési pontja 12](#_Toc133846910)

[4.2. A parser komponensen végzett munkám 12](#_Toc133846911)

[4.2.1. A default argumentumok 13](#_Toc133846912)

[4.2.2. A proxy függvény 14](#_Toc133846913)

1. Bevezetés

Szakdolgozatom témájául az Es2panda fordítóprogramon végzett fejlesztéseimet szeretném bemutatni. Mivel a fordítóprogramon végzett munkáim rendívül szerteágazóak, így sok különböző problémával találkoztam a fejlesztés során, melyekre a megoldás nem mindig volt triviális a már meglévő szoftver architektúra mellett. A fejlesztés alatt a következők voltak a céljaim:

* Robosztus könnyen érthető kód írása, amely követi a C++ nyelv bevált gyakorlatait. Ezek magukba foglalják az explicit memória kezelése minimalizálását, a const korrektséget, a makró használat minimalizálását, valamint a komponensek szoros összekapcsolódásának elkerülését.
* A futásidei teljesítmény növelése, akár a fordítási idő romlása árán. Ez leginkább a template metaprogramozási gyakorlatokkal értem el, melyek drasztikusan javítottak a kód hívó oldalának egyszerűsítésében.
* Az implementált függvények és típusok interfészeinek egyszerűsítése, hogy minél nehezebb legyen őket rosszul használni.
* A modern fordítóprogramok által alkalmazott technikák kutatása és az Es2pandába való integrálásuk. Ezek magukba foglalják a különböző optimalizációkat és statikus analíziseket.

A következő oldalakban, betekintést adok az Es2panda általam fejlesztett részeibe, azok tervezési folyamataiba, és példákat mutatok a teljesítmény orientált gondolkodásmódra, amely elengedhetetlen egy fordítóprogram esetében. Először is bemutatom az Es2pandát jelenlegi állapotában, majd egy áttekintést nyújtok az általános struktúrája felett. Bemutatom az Es2panda hat nagy komponensét és rajtuk végzett munkámat, mindezt a komponensek függőségének sorrendjében, kezdve a belépéstől a kész bytecode-ig. A komponenseket és az Es2panda kódját az általa támogatott ETS nyelvet érintő részekből fogom bemutatni. Végezetül kitérek a generált bytecode-ra és az érdekesebb implementációs részletekre.

1. Mi az az Es2panda?

Ebben a fejezetben általánosan beszélek az Es2pandáról mint szoftverről.

* 1. Mi az a fordítóprogram frontend?

A fordítóprogram frontend egy olyan szoftver, amely egy előre meghatározott nyelvet vagy nyelveket egy köztes reprezentációra vagy más néven bytecode-ra fordítja, melyet később egy fordítóprogram backend tényleges gépiódra fordít, vagy egy interpreter futásidőben értelmez.

* 1. Mi az az Es2panda?

Az Es2panda egy fejlesztés alatt álló fordítóprogram frontend melyet C++ nyelven fejlesztünk. A program több különböző szkriptnyelvet támogat és párhuzamos fordítással rendelkezik.

* 1. Hogyan fejlesztjük az Es2pandát?

Az Es2panda fejlesztése Linux platformon történik, a projekt fordításához szükséges csomagokat egy bootstrap szkript tudja telepíteni Ubuntu 20.04-es és 22.04-es kiadásokra. A build rendszer a CMake 3.10-es verziója. A build konfigurálható gcc és clang fordítóprogrammal is, a támogatott verzió gcc esetében 11.1, clang esetében pedig 14.0. Az Es2panda 64 bites Unix alapó rendszerekre lefordítható, mind x86-64 mind arm64 architektúrára. A projekt a C++17-es nyelv szabványt használja.

A verzió kezelést biztosító szoftver a git. A forráskód megtalálható a Gitee weboldalon. A különböző toolchain komponensek, mint a runtime, az Es2panda, és a runtime nyelvi pluginjai (Ecmascript, ETS) különböző repositorykban érhetők el.

1. Az Es2panda struktúrája

Az Es2panda mind logikailag, mind fizikailag szegmentálva van a struktúráját illetően. Ez a szegmentáció segít átlátni az absztrakciókat, kezelni a dependenciákat, elrejteni az implementációs részleteket és optimalizálni a fordítási időt.

* 1. A fizikai struktúra

Az Es2panda a *panda::es2panda* névtéren belül helyezkedi el, a kód példákban ezt a névteret nem fogom külön megnevezni. Az program a standard C/C++ header módszerrel van strukturálva. Az absztrakt szintaxis fa (AST) minden node-ja a saját header fájljában van, viszont a hat nagy logikai komponens javarészt monolitikusan egy-egy header fájlban található. Néhány függvény, amelyek különben implementálhatók lennének egy külön fordítási egységben (*.cpp*), mégis headerben van implementálva. Ennek az oka a fejlesztés elején a fordítási idő gyorsítása volt, valamint a hívások jobb optimalizálása fordítási egység határokon (*inlining)*. Ezeket a függvényeket a fejlesztés jelen szakaszában próbálom megszüntetni, ha egy módosításom érinti őket, ugyanis a támogatott fordítóprogramok már képesek a linkelési idejű optimalizációra (LTO), így a legtöbb belső szimbólum deduplikálásra kerül.

A projekt tartalmaz több olyan fájlt is, amelyeket a build rendszer fordítás előtt generál. Ezek a fájlok általában olyan kódot tartalmaznak, amelyek repetitívek, vagy könnyen generálhatók egyszerű markup nyelvekből. A projekt a YAML markup nyelvet használja, amit Ruby szkriptek dolgoznak fel, majd az Embedded Ruby (ERB) rendszer alakítja át megfelelő forrás fájllá, általában headerré. Néhány generált fájlra a logikai egységek részletezésénél külön ki fogok térni.

Az Es2panda két ELF állományból áll. Az egyik az *libes2panda-lib.so* programönyvtár, ami a tényleges fordítási logikát tartalmazza, a másik maga az *es2panda* nevű futtatható állomány, ami dinamikusan linekelt az előző programönyvtárhoz. Ez a szeparáció lehetővé teszi, hogy a fordítóprogram a futtatható állomány nélkül is integrálható lehessen más rendszerekbe. Azonban természetéből adódóan a szeparáció ront az LTO hatékonyságán, mivel a programkönyvtár szimbólumait nem lehet eltávolítani a dinamikus linkelés miatt.

* 1. A logikai struktúra

Ebben a fejezetben röviden bemutatom a logikai egységeket, a mélyebb működésükre és implementációs részleteikre a saját fejezeteikben fogok kitérni, kivéve az első egység esetében. Mivel ennek az egységnek a fejlesztésében nem foglaltam kulcsfontosságú szerepet, így nem szentelek neki külön fejezetet. Az Es2panda fordítóprogram hat nagy logikai elemre van strukturálva. Ezek név szerint a Lexical Analyser, a Parser, a Binder, a Semantical Checker, a Code Generator, és az Emitter. Van egy hetedik elem is, amin az előbb említett logikai elemek dolgoznak, ez az absztrakt szintaxis fa (AST).

* + 1. A Lexical Analyser (Lexer)

A Lexical Analyser (Lexer) felelős a beolvasott forráskód tokenizációjáért. Az Es2panda indulásakor a Lexer az első komponens amelyik inicializálódik. A Lexer szolgáltatja a Parser számára a tokeneket és a sor, oszlop információt hibaüzenetek esetén. Ennek a komponensnek egy része, a kulcsszavak beolvasása egy YAML markup fájlból generálódik a build során.

A Lexerből a *Lexer::NextToken* függvénnyel lehet az aktuális tokent kikérni, ez előre is mozgatja a lexer fejét a következő tokenre. Lehetőség van az aktuális pozíción lévő tokent követő karakter megvizsgálására, ezt a *Lexer::Lookahead* függvény valósítja meg. Esetenként előfordul, hogy a parsolás során a Lexert vissza kell tekerni. Erre lehetőséget adnak a *Lexer::Save* és a *Lexer::Rewind* függvények.

A generált *keywords.h* nevű fordítási egység valósítja meg a tényleges tokenizációt. A generált fájl függvényei a *Keywords::Scan\_\** alakúak, ahol a ‘*\*’* a következő betűt jelöli. Például, ha az első függvény a *Keywords::Scan\_i* volt, akkor a következő hívott függvény négy különböző lehet. Lehet a *Keywords::Scan\_if* ami ha f-et whitespace követi felvesz egy ’if’ *Lexer::Token-t* ami kulcsszó. Hasonlóan lehet a *Keywords::Scan\_im és Keywords::Scan\_in* mely függvények az ’implements’ és ’interface’ kulcsszavakat próbálják szkennelni. Ha ezek közül egyik sem teljesül, akkor a negyedik esetben az előállított *Lexer::Token* azonosító (identifier) lesz.

Az **1. Kód Részlet**: ’async’ kulcsszó YAML rekord. mutatja hogyan lehet egy kulcsszót felvenni a leíró YAML fájlba.

A *name* mező a kulcsszó konkrét formáját adja meg, amit a Lexer keresni fog a tokenizáció során. A *token* mező a *Lexer::Token-hez* tartozó *Lexer::TokenType* generált enum egyik mezőjének neve lesz. A *keyword* és *keyword\_like* mezők megadják, hogy melyik nyelvben hogyan viselkedik ez a kulcsszó. Mivel az Es2panda fordítóprogram több nyelvet is támogat ezért a Lexer nagy része újra használható a nyelvek közös pontjai között. Tehát ahogy azt **1. Kód Részlet**: ’async’ kulcsszó YAML rekord. mutatja, az *async* kulcsszó az AssemblyScript és ETS nyelvek esetén hard kulcsszó, azaz minden kontextusban kulcsszó, míg Ecmascript és Typescript nyelvek esetén soft kulcsszó, azaz a megfelelő pozícióban kulcsszó más pozícióban pedig azonosító.

keywords:

- name: 'async'

token: KEYW\_ASYNC

keyword: [as, ets]

keyword\_like: [js, ts]

1. Kód Részlet: ’async’ kulcsszó YAML rekord.

* + 1. A Parser

A Parser logikai egység a Lexer által előállított tokenekből építi fel az absztrakt szintaxis fát. Mivel az implementációja egy LR(1) -es parser így ezt alulról felfelé teszi, először a levél node-okat állítja elő majd azok szülő node-jait, így haladva egészen a gyökér node-ig. A Parser hívja a Binder megfelelő függvényeit, hogy a létrehozott AST node-ok megfelelő szkópokba legyenek bekötve.

* + 1. A Binder

A Binder komponens a Parser komponenssel együtt dolgozva építi fel a szkóp fát. A szkópok *binder::Variable* pointereket kötnek nevekhez egy *std::unordered\_map-ben*. A *binder::Variable* igazából egy szimbólum mivel nem csak változóra mutathat, hanem típus névre is és a megfelelő AST node deklarációjára mutat. Minden egyes szkópban a feloldás úgy történik egyszerűbb esetben, hogy a szkóp fát alulról felfelé bejárjuk, mindig csak a szülőkön iterálva, és megkeressük név alapján a megfelelő *binder::Variable-t*.

* + 1. A Semantical Checker

A Semantical Checker az Es2panda legkomplexebb logikai egysége. Feladata a kód szemantikai analízise. Ez Ecmascript esetében gyakorlatilag egy üres operáció, minden más támogatott nyelv esetén viszont több feladatot is el kell látnia. Az Es2panda által támogatott nyelvek az Ecmascript kivételével mind rendelkeznek valamilyen típus rendszerrel. A típusok által közölt információból és a nyelvek által támogatott különböző programozási paradigmák szabályrendszeréből a fordítóprogramnak warning és error üzeneteket kell előállítania, ha valamilyen szemantikai szabály sértett. A Semantical Checker gyakorlatilag majdnem minden AST node-hoz rendel egy *checker::Type* példányt, amelyben leírja az adott típust és különböző típus relációkat tud felállítani. Például osztályok között inheritencia láncot tud felállítani, vagy primitív típusok közötti konverziókat tud a primitívek mérete alapján engedélyezni vagy letiltani.

* + 1. A Code Generator

A Code Generator az absztrakt szintaxis fából, és a Semantical Checker által előállított típus információból generál bytecode-ot. Ez a bytecode statikusan típusos, kivéve Ecmascript esetén, ahol a runtime egy plugin segítségével támogatja a dinamikus bytecode-ot. A Code Generator párhuzamosan fut a fordítási egységekben lévő függvényekre. Ezt a működést mutexek nélkül tudja végezni (kivéve a szál scheduling esetében). A bytecode gyakorlatilag egy regiszter alapú veremgép. Egy regiszter az akkumulátor szerepét tölti be. A Code Generator felelős a regiszterek megfelelő kiválasztásáért a regiszter spillelésért. A Code Generator egy vectorban tárolja el a kigenerált utasításokat.

* + 1. Az Emitter

Az Emitter komponens a kigenerált utasítás vectort használja fel. Gyakorlatilag metainformációt ad hozzá a bytecode-hoz. Ez osztályok esetén lehet például, hogy melyik másik osztályból öröklődik, mezők és osztály függvények esetén lehet a láthatóság. A generikus osztályok és függvények is annotálva vannak a típus paramétereik számával. Ezeknek az információknak egy részét a runtime más részét a debugger használja fel.

* + 1. Az Absztrakt szintaxis fa (AST)

Az AST az Es2panda legfontosabb adat szerkezete. Megvalósítását tekintve egy osztály hierarchiából áll, melynek gyökere az *ir::AstNode* pure virtual base osztály. Ez az osztály három fontos adattagot deklarál. Az első az *ir::AstNodeType* enum, aminek segítségével egy makrón keresztül *Is##NodeType* és *As##NodeType* típus ellenőrző, és típus konverziós függvények kerülnek kigenerálásra. A második szintén egy enum típusú adattag, az *ir::ModifierFlags*. Ahogy a neve is mutatja ezek a flagek bizonyos modifier kulcsszavak meglétét tárolják egy AST node-on. Ilyenek például a Java-ból ismert és az ETS által is használt láthatóságot szabályozó kulcsszavak a *public, protected, private* vagy más egyéb módosítók például *abstract, async, const, constructor, static*. A harmadik legfontosabb adattag a szülő node-ra mutató AST node pointer.

Az *ir::AstNode* négy fontos pure virtual függvényt deklarál. Ezek az *Iterate, Dump, Check* és *Compile* függvények. A *Check* és *Compile* függvények külön overloadokkal rendelkeznek JavaScript és ETS esetekre.

* Az *ir::AstNode::Iterate* virtuális függvény paraméterében egy callback függvényt vár, amelynek szignatúrája *void(ir::AstNode \*)*. Ez a virtuális függvény egy node minden gyerekére meghívja a paraméterben kapott callback függvényt.
* Az *ir::AstNode::Dump* kírja az adott node JSON reprezentációját. A függvényt az előzővel együtt használja a program a teljes AST kiírására. A függvény paramétere egy *ir::AstDumper* pointer. Ez az osztály képes több különböző primitív, vagy tömb szerű adatszerkezetet JSON formátumba alakítani.
* Az *ir::AstNode::Check* függvény visszaadja egy adott node típusát. A típust a Checker komponens állítja elő. Általában a függvény mellékhatása, hogy a node típusát is beállítja, ha a node egy *ir::TypedAstNode*.
* Az *ir::AstNode::Compile* függvényt a Code Generator hívja. Ebben a függvényben az AST node elvégzi a lefordításához szükséges lépéseket, regisztereket, esetleg labelöket allokál majd az eredményt az akkumulátorba teszi.

Az *ir::AstNode-nak* három fontosabb leszármazott osztálya van. Az első az *ir::Statement* ami az utasítások ősosztálya. A második az *ir::TypedAstNode* amely kibővíti a funkcionalitást típus információval. A másodikból örököl a harmadik *ir::Expression*, ami a kifejezések ősosztálya.

1. A Parser

A Parser komponens állítja elő az AST-t. Az Es2panda által használt parser egy LR(1) parser. Ez azt jelenti, hogy az AST-t alulról felfelé építi fel. A Parser komponens *parser::AllocNode* függvénye az *ir::AstNode::Iterate* függvény mechanizmusát használja az új node-ok létrehozására. Miután lefut az allokálni kívánt node konstruktora, a *parser::AllocNode* meghívja az *ir::AstNode::Iterate* függvényt egy olyan callbackkel, amely az allokált node összes gyerekének beállítja a szülőjét az új node-ra. Ez gyakorlatilag a Parser általános mechanizmusa.

* 1. A Parser belépési pontja

A Parser belépési pontja a *parser::Parser::ParseProgram* függvény. Ez a függvény ETS esetében először is felépíti az ETSGlobal nevű osztályt. Erre azért van szükség mert a bytecode nem támogatja a globális változókat és függvényeket, viszont az ETS nyelv igen, erre nyújt megoldást az ETSGlobal. Ez az osztály gyakorlatilag egy shadow osztály, amelyre minden globális változó és függvény fel lesz véve, mint statikus property.

Következő lépésben a default import-ok kerülnek parsolásra. Erre azért van szükség, mert az ETS nyelv előírja, hogy minden fordítási egység impliciten importálja az *std.core* csomagot. Ebben a csomagban érhető el a Java-hoz hasonlóan az *std.core.Object* osztály, ami az osztály hierarchia gyökere. Ezek mellett még sok más funkcionalitást is biztosít.

Az utolsó lépésben a *parser::ETSParser::ParseETSGlobalScript* függvény hívódik. Ez a függvény először összegyűjti az egymodulba tartozó fájlokat, majd elindítja a parsolás lényegi részét a *parser::Parser::ParseTopLevelDeclaration* függvényt, amely addig olvassa a Lexer által előállított tokeneket ameddig el nem jut a *lexer::TokenType::EOS-ig*. Ezután allokálódik a AST gyökér node-ja az *ir::ETSScript*.

* 1. A parser komponensen végzett munkám

A Parser komponensen végzett munkám főként a default és named argumentumok parsolása. Ezek az ETS nyelvi elemek a Python-hoz hasonlóan lehetővé teszik, hogy egy függvényt opcionális argumentumokkal ruházzunk fel és az is lehetséges, hogy az argumentumokat a függvény deklaráció argumentum listájától eltérő sorrendben adjuk át, expliciten megnevezve azokat.

* + 1. A default argumentumok

A default argumentumokkal rendelkező függvények esetében nem csak egy, hanem kettő függvényt is elő kell állítani. Az eredeti függvény mellett a második egy wrapper függvény, aminek feladata, hogy runtime megvizsgálja, hogy ténylegesen mely argumentumok lettek átadva, és a hiányzó argumentumok default incializáló expressionjét meghívja.

Az első feladat létrehozni egy új AST node-ot. Ez az AST node az ir::*ETSParameterExpression* tartalmazza az információt minden függvény, minden argumentumáról. Gyakorlatilag minden *ir::ETSParameterExpression* két részből áll, amit az ETS nyelv nyelvtana diktál. Az egyik egy *ir::Identifier*, a másik pedig egy *ir::Expression*. Ezek közül a második az argumentum default initializere, ami opcionális. A **2. Kód Részlet**: **Az *ir::ETSParameterExpression* struktúrája** mutatja az AST node struktúráját.

namespace panda::es2panda::ir {

class ETSParameterExpression : public Expression {

public:

    explicit ETSParameterExpression(Identifier \*left, Expression \*right);

private:

    Identifier \*name\_;

    Expression \*initializer\_;

};

}  // namespace panda::es2panda::ir

2. Kód Részlet: Az *ir::ETSParameterExpression* struktúrája

A getter függvényeket a kód részlet tömörségének érdekében kihagytam. Setter függvényekkel a node nem rendelkezik, mivel sem egy argumentum neve, sem default initializere nem változhat.

A következő feladat az új node parsolása. A default argumentum parsolása a meglévő parser infrastruktúrával egyszerű. Először is egy nevet, azaz *ir::Identifier-t* kell parsolni majd egy típus annotációt. Ha ezen a ponton hiányzik bármelyik is akkor szintaxis hibát dobok. Ezek után opcionálisan parsolni kell egy *ir::Expression-t*, ha az argumentumnak van default initializere. Ezt a lépés sorozatot addig kell ismételni ameddig van még argumentum a függvény deklaráció szignatúrájában.

* + 1. A proxy függvény

Miután a teljes függvényt beparsoltam, elő kell állítani egy proxy függvényt, amely becsomagolja azokat a függvény hívásokat, amelyek használják a default argumentumok értékét. Egy default argumentum initializerét természetesen csak akkor kell kiértékelni, ha azt az argumentumot nem adják át a függvénynek. Ahhoz, hogy ez eldönthető legyen a függvény argumentumai közé be kellen szúrni még egy flaget amit a compiler állít be, hogy ténylegesen lett-e passzolva argumentum, ezt a flaget megvizsgálni és annak alapján kiértékelni a default initializert. Ez a logika rontana a függvény futás idején abban az esetben, amikor nincs használva default initializer. Ezért van szükség a proxy függvényre, amely pontosan ezt a logikát implementálja.

* + - 1. A proxy függvény struktúrája

A proxy függvény argumentum listája megegyezik a sima függvényével kivéve, hogy tartalmazza a fentebb említett flaget. A flag egy 32 bites változó, tehát 32 default argumentumot tud kódolni. Ha több default argumentumra van szükség, akkor egy újabb 32 bites flag változó kerül az argumentum listába. Miután a proxy függvény letesztelte a default argumentumokat és beállította az értékeket, áthív a tényleges overhead nélküli függvénybe. A proxy függvény minden szintetikusan előállított *ir::Identifier-e* mangled, tehát olyan név, amelyet a felhasználó sohasem tud leírni. Ez gyakorlatilag azt jelenti, hogy minden ilyen névbe egy ’#’ karakter van belefűzve. A compiler csak olyan hívásokat cserél a proxy hívására, ahol szükséges legalább egy default argumentum használata.

* + - 1. A proxy függvény parsolása

A proxy függvény parsolása természetesen nem tud úgy történni, mint egy sima függvény parsolása, mivel nincs jelen a forráskódban. Maga a proxy függvény létrehozásához nincs szükség semmilyen szkóp, vagy típus információra, ezért az egész szintetikus függvény létrehozható parse időben. Mivel kézzel felépíteni egy teljes függvény AST-jét komplex fenntarthatatlan kódhoz vezetne, ezért helyette a *parser::InnerSourceParser-t* használom. Ez az osztály gyakorlatilag kicseréli a Lexer komponenst az aktuális Parser alatt, és egy előre összeállított stringre cseréli a forráskódot. A parsolás tehát úgy történik, hogy a proxy függvény ETS kódját előállítom egy *std::string-be* majd a Parser mintha csak egy sima függvényt látna, beparsolja és a Binder megfelelő függvényeivel létrehozza a szkópokat és beköti a változókat.

3. Kód Részlet: InnerSourceParser indítása

const InnerSourceParser isp(this);

GetContext().Status() |= ParserStatus::ALLOW\_HASH\_MARK;

const auto lexer = InitLexer({"<default\_methods>.ets", proxy\_method});

auto \*const ident = AllocNode<ir::Identifier>(util::StringView(proxy\_method\_name), Allocator());

ParseClassMethodDefinition(ident, method->Modifiers());

GetContext().Status() &= ~ParserStatus::ALLOW\_HASH\_MARK;

## Nyilatkozat

Alulírott Szűcs Dániel mérnökinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, mérnökinformatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

2023.

Szűcs Dániel