Dokumentace projektu

Implementace překladače imperativního jazyka IFJ24

Tým **xkalinj00**, varianta **TRP-izp**

Implementovaná rozšíření:	**Jméno a příjmení:**	**Login:**
FUNEXP	Jan Kalina (vedoucí)	xkalinj00
Vlastní rozšíření viz 7. Závěr	David Krejčí	xkrejc00
	Lukáš Farkašovský	xfarkal00
	Pavel Hýža	xhyzapa00

Středa 4. prosince 2024

Obsah

1 Úvod

1.1 Členění projektu

Textová část dokumentace přesahuje doporučený rozsah stran z 5 na 8. Je to kvůli tomu, že jsme se dočetli v FAQ předmětu, že v případě rozsáhlého projektu je toto tolerováno. Na projektu jsme všichni strávili ohromné množství času a práce, tak jsme si dovolili limit 5 stran mírně překročit, abychom mohli lépe vykomunikovat naši práci.

Implementace jednotlivých částí překladače v souborech:

Lexikální analýza: scanner.[c|h]

Syntaktická analýza: parser_common.[c|h], llparser.[c|h], lltable.[c|h], precedence_parser.[c|h], precedence_stack.[c|h], precedence_table.[c|h], ast_interface.[c|h], ast_nodes.h

Sémantická analýza: semantic_analyzer.[c|h]

Generování kódu: tac_generator.[c|h], built_in_functions.[c|h]

Další: dynamic_string.[c|h], frame_stack.[c|h], symtable.[c|h], error.[c|h], ifj24_compiler.[c|h], Makefile, Doxyfile

1.2 Práce v týmu

Jako tým jsme se scházeli osobně průměrně jednou týdně ve studovnách FITu, průměrná délka sezení byla 3 hodiny. Pro distanční komunikaci jsme použili Discord, založili jsme si vlastní server s textovými i hlasovými kanály a vlákny pro organizovanou komunikaci. Jako prostředí pro samotný projekt jsme použili GitHub a Visual Studio Code. Následují vlastní slova členů týmu, o tom, na čem všem pracovali během projektu.

Jan Kalina – vedoucí týmu, chybové stavy, úprava dynamických řetězců, komunikace modulů překladače, syntaktická analýza (precedenční i LL), návrh a implementace AST, LL gramatika a LL tabulka, precedenční tabulka a redukční pravidla, syntaktické testy, testy AST, integrační testy, testovací programy, vizualizace AST, Makefile, Doxyfile, konfigurační soubory VS Code, organizace týmu, plánování termínů a schůzek, správa GitHub repositáře, kontrola práce týmu, vymáhání dodržení konvencí a programové dokumentace.

David Krejčí – sémantická analýza, generování kódu, lexikální testy, sémantické testy, testy generování kódu, testovací programy, automatický generátor LL tabulky, úprava dynamických řetězců, úprava lexikálního analyzátoru, návrh rozhraní mezi syntaktickou a sémantickou analýzou.

Lukáš Farkašovský – správa Discord serveru, dynamický řetězec, testy pro tabulku symbolů, lexikální testy, testy pro generování kódu, úprava lexikálního analyzátoru, shrnutí dotazů z fóra a Discordu, opravy chyb v různých souborech.

Pavel Hýža – lexikální analyzátor, lexikální testy, testovací programy, vestavěné funkce generování kódu, sepisování dokumentace a sjednocování stylu dokumentací členů týmu, příprava obecné osnovy pro obhajobu.

K přerozdělení bodů došlo po domluvě v týmu. *Bylo to, protože já, Pavel, mám poměrně velké mezery v programování C a ostatní členové týmu mě často vedli a pomáhali mi se základy, abych splnil zadanou práci, a to v daném standardu. Souhlasím s tím, že David a Jan získají část mého hodnocení jako „odměnu“.*

2 Lexikální analýza – Scanner

Scanner disponuje funkcí, která se stará o **třídění vstupních znaků do 7 námi definovaných skupin**: písmena, číslice, bílé znaky, jednoduché operátory, složité operátory, znaky mimo jazyk IFJ24 a znak EOF. Detailní dělení znaků a tokenů je v přílohách 8.1. a 8.2. Díky zavedení skupiny znaků, co nepatří do jazyka IFJ24, je s nimi možné psát komentáře nebo je vkládat do znakových řetězců a například připojovat jejich hodnoty pomocí escape sekvence \x(HEXA)(HEXA).

Konečný automat lexikálního analyzátoru je definován v příloze 8.3. Při načítání písmenných, číslicových i složitých řetězců může dojít k tomu, že načtený znak již nepatří do stávajícího tokenu, ale do dalšího. O tyto případy se stará funkce, která vrátí daný znak zpět na vstup, aby po resetu FSM mohl být načten znovu.

Při načítání řetězců s písmeny se po ukončení řetězce volá funkce, která zjišťuje, zda se jedná o identifikátor, nebo jedno z klíčových slov. Skupina znaků jednoduché operátory je definována tak, že hned po obdržení znaku je znám token. Je tak v rámci efektivity oddělena od skupiny složité operátory. Pro uložení obsahu identifikátorových, číselných a řetězcových tokenů jsou **využívány dynamicky alokované řetězce znaků**. V případě zjištění klíčového slova, zůstane řetězec nevyužit, proto se před ukončením chodu FSM uvolní. Při dosažení znaku konce souboru EOF se scanner cyklí, dokud to od něj vyžaduje syntaktický analyzátor.

3 Syntaktická analýza – Parser

3.1 LL gramatika a LL tabulka

Realizace syntaktického analyzátoru začala návrhem LL gramatiky pro parsing. Na papíře byl vytvořen **seznam řídících konstrukcí jazyka**, způsobů deklarací proměnných a volání funkcí s jejich strukturami. Následně byly pro tyto struktury postupně definovány neterminály, které byly často dále rozdělovány na menší části. Například v neterminálu `<PARAMETERS>` pro parametry funkce byla levá rekurze, kterou bylo žádoucí odstranit, proto vznikl dodatečný neterminál `<PARAM_LIST_REST>`.

Pro tvorbu **množin** `EMPTY`, `FIRST`, `FOLLOW` a `PREDICT` byl vytvořen **C++ program**, který je uměl **automaticky generovat**, čímž se zjednodušila implementace změn v gramatice v průběhu času.

Při realizaci **LL tabulky** jsme uvažovali nad řešením pomocí dvourozměrného pole, avšak rozhodli jsme se pro implementaci **vlastní struktury**. Ta obsahuje 2 parametry, klíč (neboli terminál, který označuje řádek tabulky) a pole LL pravidel. Řádek tabulky odpovídá jednotlivým terminálům, zatímco výčet neterminálů reprezentuje indexy do pole LL pravidel. Souřadnice v tabulce jsou dány dvojicí [Terminál, Neterminál], podle kterých se najde potřebné LL pravidlo. Vyhledávání v tabulce je realizováno binárně, což je efektivní a rychlé.

3.2 AST (Abstraktní syntaktický strom)

Navrhli jsme abstraktní syntaktický strom. Stanovili jsme si cíl, že AST bude možno **sestavovat průběžně** během rekurzivního sestupu při LL parsingu. Zároveň jsme se rozhodli namísto jednoho univerzálního uzlu vytvořit minimální počet různých uzlů kvůli přehlednosti,

tak jsme došli k závěru používat ukazatele na `void`, namísto unií, jejichž obsah by prospěl spíše univerzálním uzlům. S ukazatelem na `void` lze pak svázat libovolný uzel.

V prvním návrhu měl každý neterminál svůj vlastní typ uzlu. Později jsme prováděli **sjednocení**, například uzel pro literál a identifikátor, protože jsou to nejzákladnější terminály všech výrazů. Potom jsme vytvořili uzly pro konstrukce `if-else` nebo cyklus `while`, které obsahují ukazatele na své části jako jsou například podmínka nebo tělo cyklu. Sjednotili jsme uzly pro argumenty volání funkcí a parametry definice funkcí, protože oba obsahují seznam argumentů nebo parametrů. Do uzlu příkazů jsme museli zakomponovat lineárně vázaný seznam. Právě ten tvoří těla všech funkcí a řídích konstrukcí.

Některé uzly mají jeden ze svých členů **ukazatel na `void`**, například, aby bylo možné k uzlu pro výraz připojit uzel pro identifikátor/literál nebo uzel pro volání funkce. Způsob dereference tohoto ukazatele je dán uzlovým členem, jenž určuje typ uzlu, který je s ním svázaný.

Pro jednoduché vytváření a uvolňování uzlů během syntaktické analýzy bylo vytvořeno vlastní rozhraní. Vytvářecí funkce vrací ukazatel typu `void`, jenž je nutné přetypovat (podobně jako u funkce `malloc`). Uvolňovací funkce vrací ukazatel na konkrétní uzel a podle jeho typu volá specifickou funkci. Abychom nemuseli ručně inicializovat a propojovat uzly AST během rekurzivního sestupu, byly vytvořeny funkce pro inicializace jednotlivých uzlů.

3.3 Parser Common

Cílem je minimalizovat duplicitní kód mezi různými částmi parseru. Poskytuje jednotné zázemí pro LL i precedenční parser při práci s tokeny, chybovými stavy a datovými strukturami. Zajišťuje konzistentní předávání dat napříč překladačem. V rámci syntaktické analýzy jsou používány globální proměnné: `currentTerminal` reprezentující aktuálně přijatý token jako LL i precedenční terminál, `precStackList` sloužící jako zásobník všech aktuálně využívaných precedenčních zásobníků.

Základem je správa aktuálního terminálu, který je reprezentován vlastní strukturou zahrnující samostatný typ pro oba parsery a hodnotu tokenu v dynamickém řetězci. Abychom nepřidávali dodatečné globální proměnné, Look-ahead terminál je uchováván pomocí statické proměnné uvnitř funkce, která získává aktuální token ze scanneru. Stejně se řeší i chybové stavy. Funkce `parser_errorWatcher` kontroluje chyby během parsingu, mohou být lexikální, syntaktické a sémantické, nebo se také může jednat o interní chyby překladače.

Po velkých problémech s úniky paměti kvůli chybovým stavům při 1. pokusném odevzdání byl implementován speciální mechanismus, který se stará o uchování příznaků jednotlivých typů chyb a prvního chybového stavu. Uvolnění všech alokovaných zdrojů probíhá až po propagováním chyby zpět do vstupní funkce parseru, aby nedošlo ke ztrátě dat. Uzly obou parserů jsou uvolňovány během návratu z rekurze funkcemi, které je vytvořily.

Obsahuje funkce pro mapování datových typů a tokenů, které převádějí typy tokenů získané ze scanneru do tvaru vhodného pro syntaktickou analýzu nebo z typů v AST na typy v tabulce symbolů. Tímto se udržuje konzistence mezi různými částmi překladače.

3.4 LL Parser

Rekurzivní sestup pro LL syntaktickou analýzu jsme zvolili kvůli možnosti konstruovat AST během návratu z rekurze zdola nahoru. Snažili jsme se o co nejmenší počet rozsáhlých `switch-cases` v rámci funkcí. Je prvním volaným modulem ve funkci `main()`. Žádostmi řídí scanner a propůjčuje řízení precedenčnímu parseru pro zpracování výrazů. Oba parsery během parsingu plní tabulku symbolů. V původní verzi docházelo k vytváření zbytečných rámců, ale se současnou gramatikou parser již dokáže efektivně určit, kdy je vhodné nové rámce vytvářet.

Přidává identifikátory proměnných do tabulky symbolů konkrétního rámce, kde byla proměnná deklarována. Rozhodli jsme se, že při deklaraci funkce přidá do tabulky symbolů její identifikátor, počet parametrů a informace o nich a následně přidá tyto parametry jako lokální proměnné do rámce pro tělo funkce. Při deklaraci proměnných se nejdříve kontroluje zastínění. Při parsingu použití proměnné probíhá kontrola, zda je deklarovaná v rámci správného rozsahu platnosti. Zároveň uchovává data o modifikaci proměnných a vytváření uzlů pro sémantickou analýzu. Bylo složité zajistit správnou propagaci chybových stavů všemi úrovněmi rekurze, proto byla vytvořena kaskáda návěští, která minimalizuje opakující se kód.

3.5 Precedenční tabulka

Precedenční tabulka byla navržena už od počátku tak, aby bylo možné realizovat **rozšíření FUNEXP**, proto kromě aritmetický a relačních operátorů, identifikátorů a literálů obsahuje také priority pro parsing libovolně vnořených funkcí. Byla rozšířena o určení priority tečky, klíčového slova „ifj“ a čárku, která slouží jako oddělovač argumentů při volání funkce. „ifj“ má nižší prioritu a její obdržení při parsingu vždy znamená syntax error, nebo provedení posunu. Tečka je povolena jen, když je na vrcholu zásobníku „ifj“. I zde je použito binární vyhledávání.

3.6 Precedenční zásobník

Důvodem implementace precedenčního zásobníku byla možnost v rámci aplikace redukčních pravidel průběžně stavět podstromy AST pro parsovaný výraz. Kvůli potřebě na zásobníku uchovávat mnoho rozlišných informací byla vytvořena speciální struktura pro zásobníkovou položku s názvem `PrecStacNode`. Ta obsahuje informace o tom, zda je symbol na zásobníku precedenčním terminálem, neterminálem nebo symbolem „handle“. Pokud se jedná o terminál obsahuje také informaci o tom, o který terminálový symbol se přesně jedná. Zásobníkovými neterminály jsou neterminály `<EXPR>` pro výraz a `<ARG_LIST>` pro seznam argumentů volané funkce. Pokud se jedná o neterminál je v uzlu uveden typ AST uzlu, který byl pro tento neterminál vytvořen, tento uzel je svázán se zásobníkovou položkou pomocí ukazatele na typ `void`. Položka se speciálním symbolem "handle" slouží jako zarážka při výběru symbolů, nad kterými bude vybíráno a následně aplikováno precedenční redukční pravidlo.

Pro rozšíření FUNEXP bylo nutné zajistit parsing možných výrazů v argumentech funkcí pomocí precedenčního parseru. Zvolili jsme přístup kombinování rekurzí pravidla LL parseru pro neterminál `<ARGUMENTS>` s precedenčním parserem. Argumenty funkce po jednom parsujeme a postupně spojujeme do seznamu argumentů funkce. Při potřebě parsovat argumenty vnořené funkce se vytvoří nový precedenční zásobník pro seznam argumentů této

funkce. Pro uchovávání precedenčních zásobníků byl vytvořen „zásobník precedenčních zásobníků“. Vždy při vytvoření nového kontextu precedenčního parseru je na tento zásobník poslán (`push`) nový precedenční zásobník a při ukončení kontextu je uvolněn (`pop`).

3.7 Precedenční parser

Přebírá řízení nad syntaktickou analýzou v rámci parsingu určitých neterminálů. Při předání řízení je mu předán i typ neterminálu, podle kterého se určí, jaký příchozí terminál mapovat na pseudo terminál `dolar`, jenž funguje jako počáteční znak na zásobníku. Například při parsingu LHS příkazu přiřazení je určeno, že pokud bude aktuálním terminálem středník, tak by měla být precedenční syntaktická analýza ukončena, aplikovat se zbývající redukční pravidla a následně by mělo být vráceno řízení. Využívá vlastní počítadlo zanoření závorek pro komplexní výrazy. Využívá se i v některých kontextech jako dodatečná podmínka mapování vstupního terminálu.

Při redukci je nejdříve zvoleno správné redukční pravidlo následovně. Existuje tabulka redukční pravidel implementována podobně jako LL tabulka, v ní je pro každé pravidlo uvedeno pole obsahující sekvenci zásobníkových symbolů až po symbol „handle“. Během výběru je zásobník procházen od vrcholu po nalezení symbolu „handle“, nebo po projití počtu symbolů odpovídající nejdelší sekvenci (5 zásobníkových symbolů). Do pole se zaznamenává sekvence navštívených symbolů, ta se následně porovná se sekvencemi v tabulce redukčních pravidel. Pokud dojde ke shodě, aplikuje se redukční pravidlo, jinak dojde k syntaktické chybě. Seznam redukčních pravidel je v příloze.

4 Sémantická analýza

Sémantická analýza má 4 fáze:

- Do rámců se přidávají položky, kontrolují se redefinice proměnných a nedefinované proměnné (probíhá během parsingu).
- Kontrola základních požadavků programu, zda obsahuje funkci `main` a správnou formu prologu (probíhá po parsingu).
- Kontrola těl funkcí, jejich příkazů a výrazů. Probíhá rekurzivním procházením AST ve stylu post-order pomocí funkcí pro jednotlivé typy příkazů a výrazů.
- Prohledání všech rámců programu, kontrola využití proměnných a neměnnost konstant.

Různé komplikace byly například, že funkce `ifj.write` a `ifj.string` mohou přijímat více typů parametrů, proto jsou kontrolovány vlastními funkcemi. Identifikátory jsou v AST uloženy bez prefixu „ifj.“, ale v tabulce symbolů prefix mají. Při volání je nutné kontrolovat, zda je funkce vestavěná a případně pro vyhledání v tabulce prefix přidat. Pro kompatibilitu typů z různých částí překladače je implementován další typ společně s funkcemi pro konverze. Kromě chyb daných zadáním se vrací chyba č. 10 (ostatní sémantické chyby) v případech, že je nevyhovující prolog, výraz zahazuje hodnotu nebo se za příkazem `return` nachází „mrtvý“ kód.

Mezi rozšíření patří:

- kontrola, že se příkaz return u funkcí s návratovou hodnotou vyskytuje v každé větvi těla funkce,

- detekce „mrtvého“ kódu za příkazem return,
- implicitní konverze při přiřazení,
- vyhodnocování výrazů se známou hodnotou při překladu.

Vyhodnocování výrazů při překladu probíhá následovně. Pokud je známá hodnota proměnné, v AST se nahradí literálem. Jestli analyzujeme binární operaci nad literály, provedeme ji, vypočítáme hodnotu a binární operaci nahradíme literálem.

5 Generování kódu

Algoritmus prochází AST stejným stylem jako sémantický analyzátor. Většina instrukcí se přímo vypisuje na `STDOUT`. Instrukce `PUCHS` a `POPS` se nejdříve předají do optimalizačního bufferu, který se je pokusí nahradit instrukcí `MOVE`.

Pro zajištění unikátních identifikátorů je **proměnným v mezikódu přidán sufix** s číslem rámce, ve kterém jsou definovány. Při zanoření do `while` cyklu se nejdříve definují všechny proměnné v těle cyklu, až poté se generuje podmínka s tělem cyklu, již bez definic. Toto zamazuje redefinici proměnných v těle cyklu.

Práce s daty v mezikódu probíhá tak, že při vyhodnocování výrazů, včetně volání funkcí, je výsledná hodnota předána na vrcholu zásobníku a všechny operace dále používají zásobníkovou verzi, pokud je to možné. Pokud není, využívají se pomocné globální proměnné definované na začátku generovaného programu.

V rámci optimalizace se snažíme sekvenci instrukcí tvořící vestavěné funkce vkládat přímo do těla funkce `main()`, čímž snižujeme počet reálných volání funkcí, které by byly výpočetně náročnější. Funkce, které takto nahradit nelze jsou definovány na začátku programu. U takových funkcí vzniká komplikace, kdy parametry a proměnné uvnitř vestavěné funkce nemají své číslo rámce, proto pro ně musí platit výjimka a jsou bez sufixu.

Pro zajištění unikátních návěští má každá `if` a `while` struktura svou unikátní hodnotu. Statické proměnné unikátních hodnot se na začátku generování kódu manuálně resetují, aby nedocházelo k „pamatování“ staré hodnoty napříč spouštěnými unit testy. V naší implementaci zajišťuje `push` a odstranění rámců s parametry funkcí volaná funkce.

6 Speciální

6.1 Zásobník rámců

Účelem je uchování dat pro sémantickou analýzu a generování kódu volání funkcí. Je implementován jako lineární seznam s ukazateli na začátek (vrchol zásobníku) a konec (globální rámec). Zásobník rámců také obsahuje identifikační číslo rámce, který byl vytvořen jako poslední.

Rámec obsahuje ukazatel na tabulku symbolů, identifikační číslo, příznak, zda se jedná o rámec funkce a ukazatel na rámec níže v zásobníku. Příznak, že se jedná o funkci znamená konec vyhledávání v zásobníku. Struktura rámce je znázorněna v příloze 8.7. „Diagram struktury tabulky symbolů“.

Protože v jazyce IFJ24 **nejsou povolené globální proměnné**, tak první (globální rámec) naspodu zásobníku obsahuje pouze data funkcí a proměnné „ifj“, v ostatní rámcích jsou naopak jen data proměnných. Pole rámců dynamicky uchovává ukazatele všech rámců vytvořených během běhu programu pro snadné vyhledávání v konkrétním rámci a uvolnění paměti na konci překladu nebo při chybě.

Algoritmus pro vyhledávání vyhledává lineárně v zásobníku (seznamu), dokud nenarazí na rámec funkce, který ještě prohledá společně s globálním rámcem.

Globální proměnné: ukazatel na zásobník rámců, pole rámců. Globální rámec také uchovává data o vestavěných funkcích, které se vkládají při inicializaci zásobníku. Položky se vkládají během syntaktické analýzy společně s částí dat. Data jsou dále využívána a doplněna během sémantické analýzy. Během generování kódu se využívají data funkcí pro generování předání parametrů.

6.2 Tabulka symbolů

Skládá se z použité velikosti, alokované velikosti a ukazatele na jednorozměrné pole položek. Každá položka má svůj klíč/identifikátor, stav, příznaky a ukazatel na data. Stav naznačuje, zda na indexu položka není, nebo pokud je, tak značí typ dat. Příznaky značí, zda položka byla v programu použita nebo změněna, jestli je konstantního typu a jestli je její hodnota známá při překladu. Struktura rámce je znázorněna v příloze 8.7. „Diagram struktury tabulky symbolů“.

Data položky mohou být dvojího typu:

- pro proměnné je to hodnota datového typu odvozeného ze stavu položky
- pro funkce je to struktura dat funkcí

Data funkce mají návratový typ, identifikátor rámce pro tělo funkce, počet parametrů a ukazatel na pole parametrů. Každý parametr má svůj identifikátor a datový typ.

Mezi použité algoritmy patří hashovací funkce (konkrétně algoritmus djb2). Indexové konflikty jsou řešeny lineárním průchodem inkrementací indexu**.**

Tabulka je dynamická a rozšiřuje se při naplnění z 60 % na dvojnásobek své předchozí kapacity. Začíná na velikosti 10 položek. Při přidávání položek a rozšiřování se vytváří kopie klíčů místo předání ukazatele. Data se naopak předávají ukazatelem. Snažili jsme se o to, aby se uživatelé nemuseli zabývat uvolňováním dat, která mohou být v tabulce. Rozhodli jsme se, že bude lepší, když některá data budou jak ve stromě, tak v tabulce. V tom případě se data uvolňují pouze ve stromě, aby nedošlo ke dvojímu uvolnění.

6.3 Dynamický řetězec

Pro užitečné využití v dalších částech projektu jsme se hned na začátku rozhodli vytvořit si datový typ dynamického řetězce. Je vysoce spjatý s vytvářením jednotlivých tokenů v rámci lexikální analýzy, ale používá se například i v AST a zásobníku rámců. Je to struktura obsahující ukazatel na dynamické pole znaků, velikost alokované paměti a skutečnou délku řetězce.

Obsahuje funkce pro práci nad danými řetězci:

- `DString_compare` … porovnání dvou dynamických řetězců
- `DString_compareWithConstChar` … porovnání s konstantním řetězcem
- `DString_DStringtoConstChar` … převedení na konstantní řetězec
- `DString_free` … uvolnění ukazatele na řetězec a celé struktury
- `DString_appendChar` … přidání znaku do řetězce, je navržen tak, aby se řetězec při přesáhnutí aktuální délky automaticky zvětšil.

Domluvili jsme se tak, že **řetězec musí být vždy uvolněn volajícím**. Při převodu mezi konstantním a dynamickým řetězcem, je vždy alokováno místo v paměti, po zavolání těchto funkcí musí být později všechno alokované místo uvolněno.

6.4 Testování

V rámci ověření funkčnosti implementace projektu jsme se rozhodli intenzivně používat unit testy pomocí GoogleTest framework. Každou hlavní část překladače jsme testovali vlastními testy, tedy lexikální analýzu, syntaktický analyzátor, sémantický analyzátor a generátor kódu. Dále jsme testovali i LL tabulku, zásobník rámců, tabulku symbolů, dynamické řetězce, AST rozhraní a error knihovnu. Nakonec jsme zavedli integrační testy pro celkový chod překladače.

7 Závěr

Projekt nám všem dal hodně teoretických a praktických dovedností, ale i cenných zkušeností při práci v týmu s dalšími lidmi. Byl pro nás všechny velice náročný a nechali jsme v něm hodně nervů a energie, někdy i vzteku a slz. I když si každý z nás odnesl něco trochu jiného, tak to pro nás všechny byla nedocenitelná zkušenost do života.

Seznam vlastních rozšíření, práce nad rámec zadání a dalších specialit:

- Vlastní sémantické kontroly
- Optimalizace mezikódu
- Dynamická tabulka symbolů
- Možnost pracovat se znaky mimo základní ASCII v rámci komentářů nebo znakových řetězců
- Kontrola meze přetečení datových typů integer a double během konstrukce AST během konverze z tokenu na číselný datový typ
- Přes 150 unit testů pomocí GoogleTest framework, které pracují s téměř 400 testovacími soubory
- Program pro generování množin `EMPTY`, `FIRST`, `FOLLOW`, `PREDICT` podle LL gramatiky
- Možnost generování detailní Doxygen dokumentace
- Propagace chybových stavů parserem v rámci lexikální a syntaktické analýzy při návratu z rekurze
- Makefile umožňující stavět jednotlivé moduly separátně i integrovaně, stavět a spouštět unit testy modulů a integrační testy celého překladače, generovat pokrytí kódu, sebedokumentující Makefile

8 Přílohy

8.1 Tabulka tokenů pro lexikální analýzu

Types of Tokens

TokenType	Identifier	Keyword(separate)	Int	Float	Special (separate)	String (", \\)	EOF
String Content	abc	const	123	1.0	(	Kentucky Fried Chicken	EOF
	def123	var	456	20.245	)	Fortnite Battlepass,(NL)	
	..	i32	...	20e123'	{	I just s**t out my ass!	
		f64		20.123e123'	}	...	
		[]u8		...	|		
		?i32			.		
		?f64			,		
		?[]u8			:		
		pub			;		
		fn			+		
		void			-		
		return			*		
		null			/		
		if			=		
		else			==		
		while			!=		
		import			>=		
		_			<=		
		ifj					

8.2 Tabulka znaků pro lexikální analýzu

DEC	CHAR	DEC	CHAR	DEC	CHAR	DEC	CHAR
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

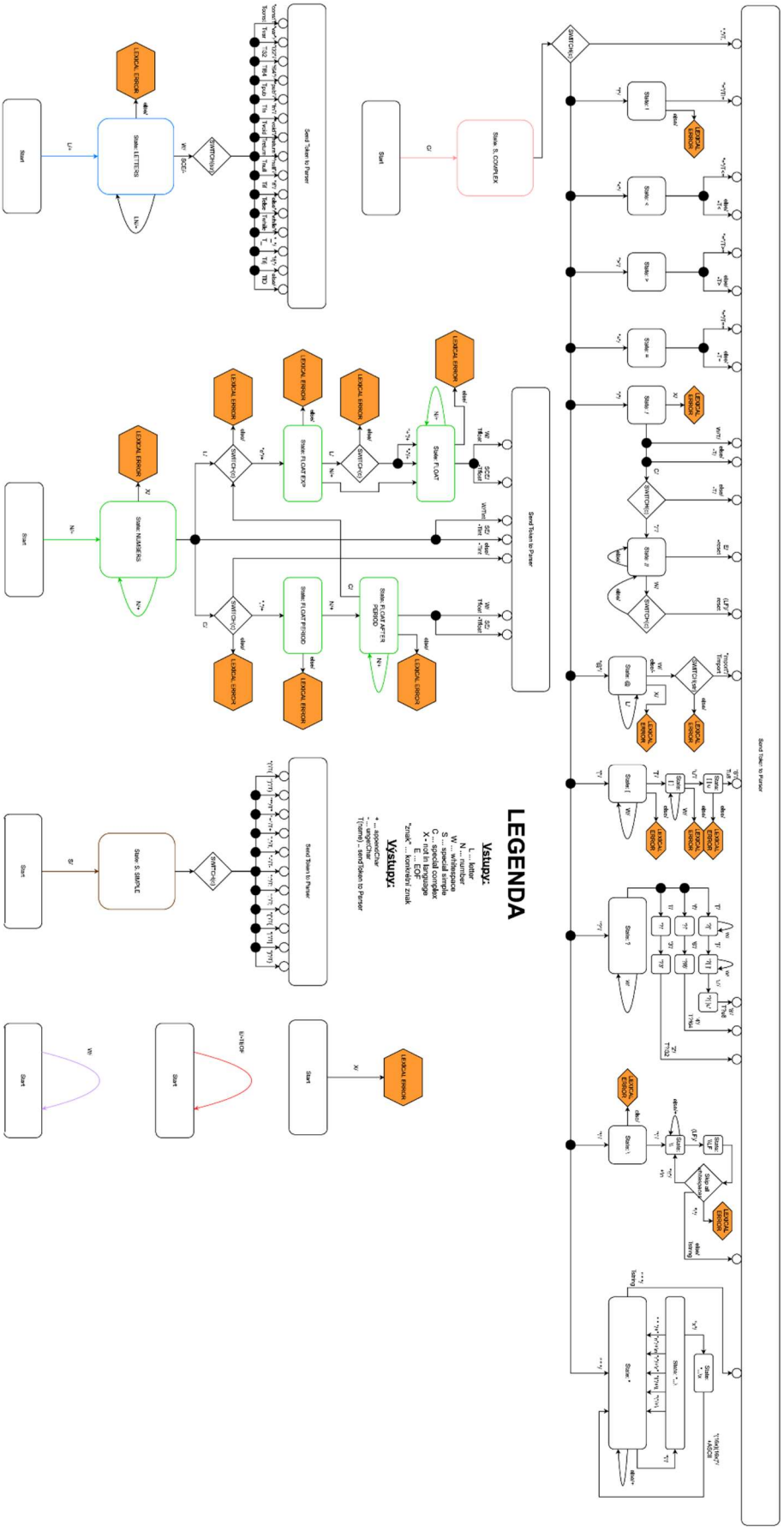
Types of Characters

DEC	CHAR
-1	EOF

DEC	Colour
0-31	(U) Unprintable
35-39, 94, 96, 126, 127+	(X) Not in language
65-90, 95, 97-122	(L) Letters
48-57	(N) Numbers
40-45, 58, 59, 123-125	(S) Special Simple
33, 34, 46, 47, 60-64, 91-93	(C) Special Complex
10, 11, 12, 13, 32	(W) Whitespace
-1	(E) EOF

Znaky vyšší než 127 lze použít například v komentářích.

8.3 Diagram konečného automatu lex. analýzy



8.4 LL gramatika

Jan Kalina, VUT FIT 2BIT, 2024/2025

GRAMATIKA PRO JAZYK IFJ24

LL-GRAMATIKA

1: <PROGRAM> → <PROLOG> <FUN_DEF_LIST> EOF

2: <PROLOG> → const ifj = @import ([prec_expr]) ;

3: <FUN_DEF_LIST> → <FUN_DEF> <FUN_DEF_LIST>

4: <FUN_DEF_LIST> → ε

5: <FUN_DEF> → pub fn id (<PARAMETERS>) <RETURN_TYPE> <SEQUENCE>

6: <PARAMETERS> → <PARAM_LIST>

7: <PARAMETERS> → ε

8: <PARAM_LIST> → <PARAM> <PARAM_LIST_REST>

9: <PARAM_LIST_REST> → , <PARAM_LIST>

10: <PARAM_LIST_REST> → ε

11: <PARAM> → id : <DATA_TYPE>

12: <RETURN_TYPE> → <DATA_TYPE>

13: <RETURN_TYPE> → void

14: <DATA_TYPE> → i32

15: <DATA_TYPE> → ?i32

16: <DATA_TYPE> → f64

17: <DATA_TYPE> → ?f64

18: <DATA_TYPE> → []u8

19: <DATA_TYPE> → ?[]u8

20: <STATEMENT_LIST> → <STATEMENT> <STATEMENT_LIST>

21: <STATEMENT_LIST> → ε

22: <STATEMENT> → <VAR_DEF> ;

23: <STATEMENT> → id <STATEMENT_REST> ;

24: <STATEMENT> → _ = <THROW_AWAY> ;

25: <STATEMENT> → <IF>

26: <STATEMENT> → <WHILE>

27: <STATEMENT> → return [prec_expr] ;

28: <STATEMENT> → ifj . id (<ARGUMENTS>) ;

29: <VAR_DEF> → <MODIFIABLE> id <POSSIBLE_TYPE> = [prec_expr]

30: <MODIFIABLE> → var

31: <MODIFIABLE> → const

32: <POSSIBLE_TYPE> → : <DATA_TYPE>

33: <POSSIBLE_TYPE> → ε

34: <STATEMENT_REST> → = [prec_expr]

35: <STATEMENT_REST> → (<ARGUMENTS>)

36: <THROW_AWAY> → [prec_expr]

37: <IF> → if ([prec_expr]) <NULL_COND> <SEQUENCE> else <SEQUENCE>

38: <NULL_COND> → | id |

39: <NULL_COND> → ε

40: <SEQUENCE> → { <STATEMENT_LIST> }

41: <WHILE> → while ([prec_expr]) <NULL_COND> <SEQUENCE>

42: <ARGUMENTS> → [prec_expr]

Poznámky:

červené ... terminály

[prec_expr] ... spec. terminál (předání kontroly precedenčnímu parseru)

<modré> ... NEterminály

ε ... epsilon (prázdný řetězec)

PRECEDENČNÍ SYNTAKTICKÁ ANALÝZA

Pravidla kombinující LL a precendenční syntaktickou analýzu:

27: <STATEMENT> → return [prec_expr] ;

29: <VAR_DEF> → <MODIFIABLE> id <POSSIBLE_TYPE> = [prec_expr]

34: <STATEMENT_REST> → = [prec_expr]

36: <THROW_AWAY> → [prec_expr]

37: <IF> → if ([prec_expr]) <NULL_COND> <SEQUENCE> else <SEQUENCE>

41: <WHILE> → while ([prec_expr]) <NULL_COND> <SEQUENCE>

42: <ARGUMENTS> → [prec_expr]

TERMINÁLY vystupující v precedenční analýze

3: **Operandy:**

- id
- i32_literal
- f64_literal
- u8_literal
- null_literal

3: **Ostatní:**

- ifj
- .
- (
-)
- $ (obecný konec vstupu)

4: **Operátory:**

a) Relační operátory:

- ==
- !=
- <
- >
- <=
- >=

b) Aditivní operátory:

- +
- –

c) Multiplikativní operátory:

- *
- /

Množnina FOLLOW aka komunikační symboly mezi parsery

FOLOW_VAR_DEF = { ; }

FOLOW_STATEMENT_REST = { ; }

FOLOW_THROW_AWAY = { ; }

FOLOW_IF = {) }

FOLOW_WHILE = {) }

FOLOW_STATEMENT = { ; }

FOLOW_ARGUMENTS = {) }

Redukční pravidla pro precedenční syntaktickou analýzu:

1: <EXPR> → id

2: <EXPR> → i32_literal

3: <EXPR> → f64_literal

4: <EXPR> → u8_literal

5: <EXPR> → null_literal

6: <EXPR> → <EXPR> + <EXPR>

7: <EXPR> → <EXPR> – <EXPR>

8: <EXPR> → <EXPR> * <EXPR>

9: <EXPR> → <EXPR> / <EXPR>

10: <EXPR> → <EXPR> == <EXPR>

11: <EXPR> → <EXPR> != <EXPR>

12: <EXPR> → <EXPR> < <EXPR>

13: <EXPR> → <EXPR> > <EXPR>

14: <EXPR> → <EXPR> <= <EXPR>

15: <EXPR> → <EXPR> >= <EXPR>

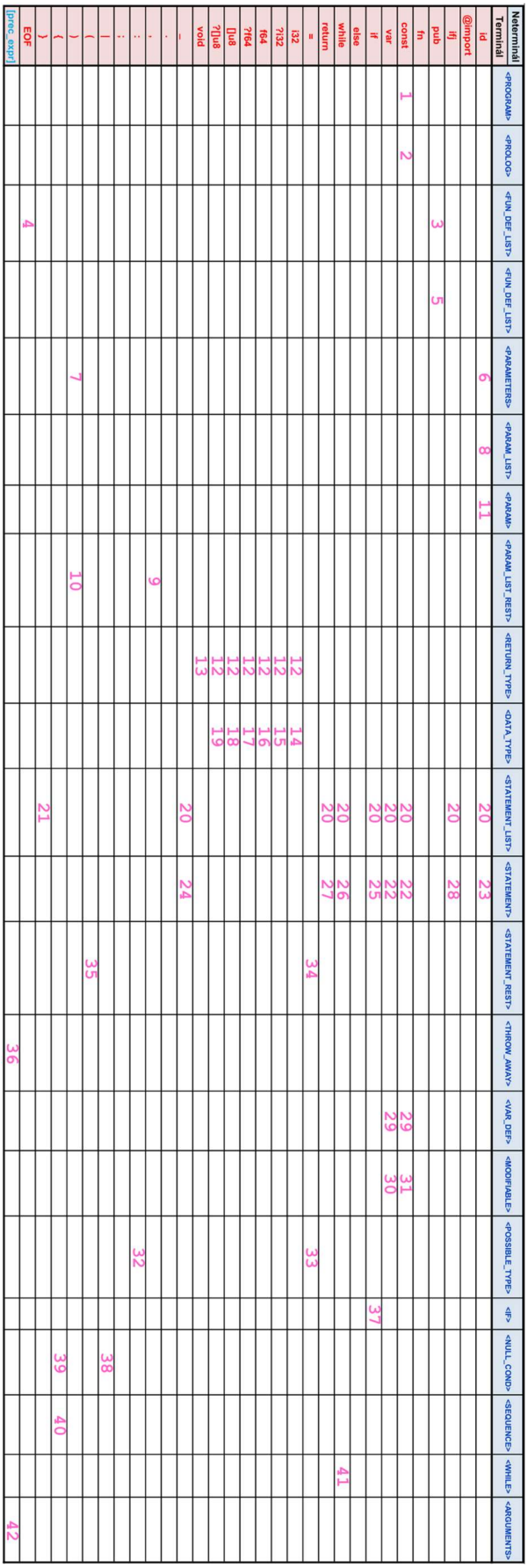
16: <EXPR> → (<EXPR>)

17: <EXPR> → id <ARG_LIST>

18: <EXPR> → ifj . id <ARG_LIST>

19: <ARG_LIST> → <EXPR> , <ARGUMENTS>

8.5 LL tabulka

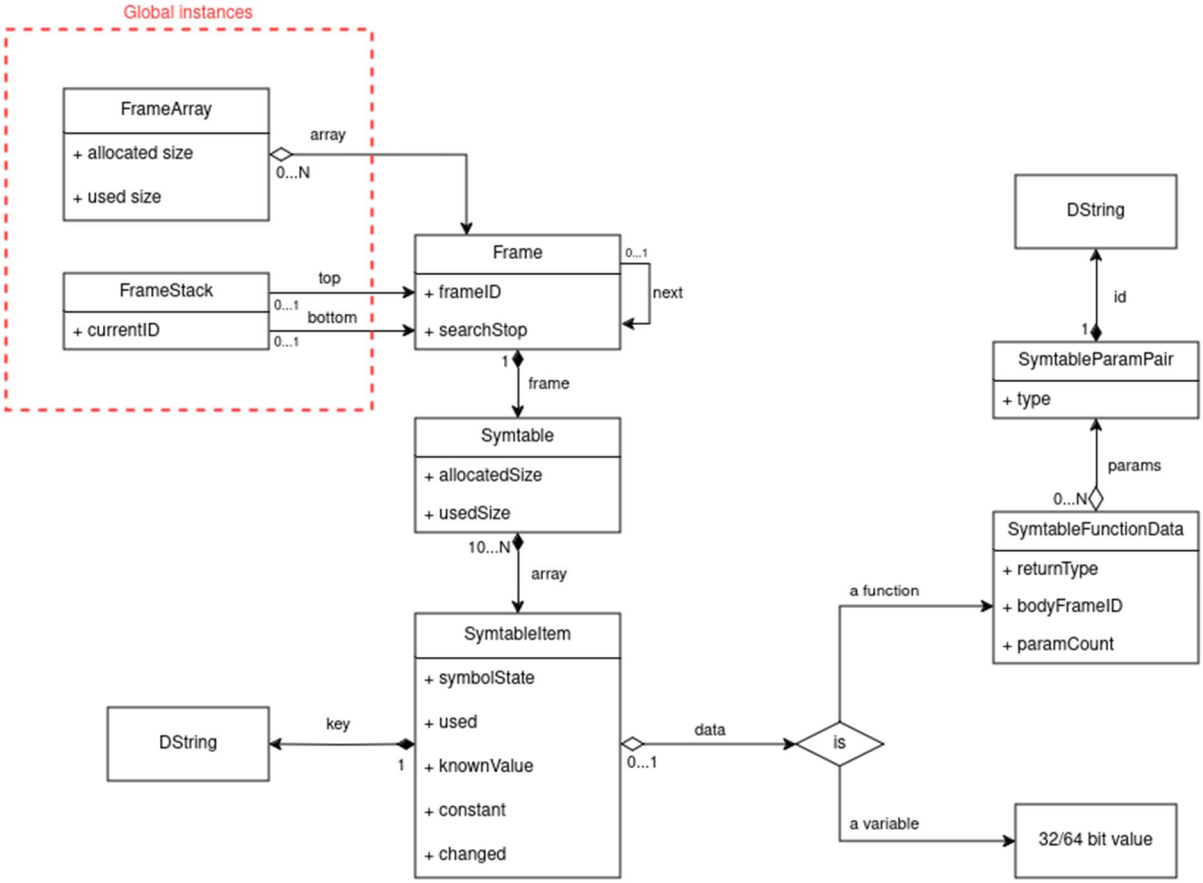


Neterminál / Terminál	<PROGRAM>	<PROLOG>	<FUN_DEF_LIST>	<FUN_DEF_LIST>	<PARAMETERS>	<PARAM_LIST>	<PARAM>	<PARAM_LIST_REST>	<RETURN_TYPE>	<DATA_TYPE>	<STATEMENT_LIST>	<STATEMENT>	<STATEMENT_REST>	<THROW_AWAY>	<VAR_DEF>	<MODIFIABLE>	<POSSIBLE_TYPE>	<IF>	<NULL_COND>	<SEQUENCE>	<WHILE>	<ARGUMENTS>
id					6		11				20	23										
@import																						
ifj											20	28										
pub			3																			
fn		2		5																		
const	1										20	22			29	31						
var											20	22			29	30						
if											20	25						37				
else																						
while											20	26								41		
return											20	27										
=									12	14			34				33					
i32									12	15												
?i32									12	16												
f64									12	17												
?f64									12	18												
[]u8									12	19												
?[]u8									12													
void									13													
_											20	24										
.								9														
,																	32					
:																						
|													35						38			
(					7			10												39		
)																						
{											21									40		
}																						
EOF			4																			
[proc_expr]														36								

8.6 Precedenční tabulka

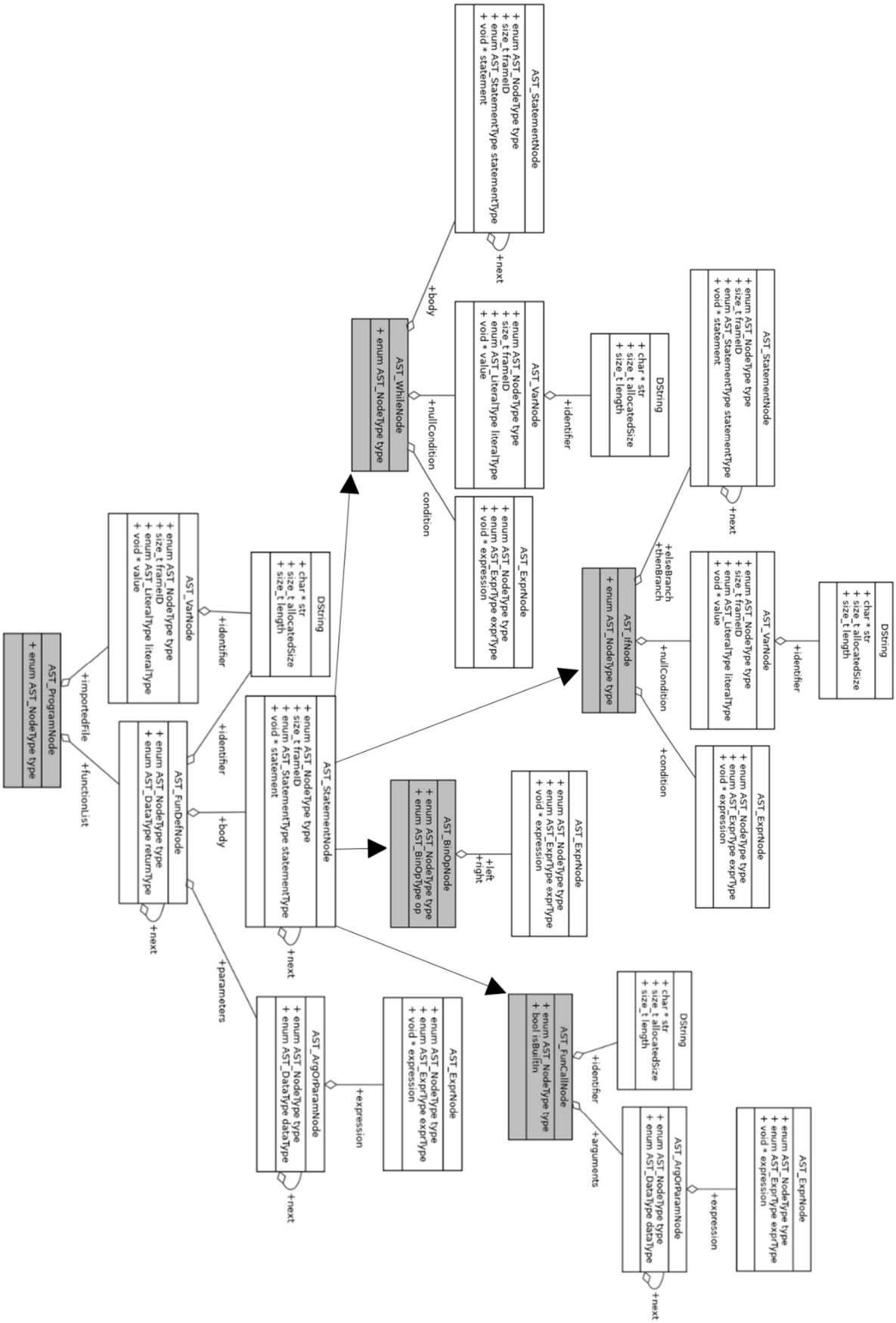
Vstupní terminál / Terminál na vrcholu zásobníku	Identifikátor	i32 literál	f64 literál	[]u8 literál	null literál	ifj	.	(	)	+	-	*	/	==	!=	<	>	<=	>=	,	$
Identifikátor																					
i32 literál																					
f64 literál																					
[]u8 literál																					
null literál																					
ifj																					
.																					
(	^																				
)	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
+	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
-	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
*	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
/	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
==	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
!=	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
<	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
>	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
<=	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
>=	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
,	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
$	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	

8.7 Diagram struktury tabulky symbolů



8.8 Příklad výstupu programu pro vizualizaci AST

8.9 Diagram struktury AST



8.10 Hierarchie projektového adresáře