Zadání projektu z předmětů IFJ a IAL

Zbyněk Křivka, Ondřej Ondryáš
e-mail: {krivka, iondryas}@fit.vut.cz

14. září 2025 (předběžná)

Obsah

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ25.
Informace: diskuzní fórum a Moodle předmětu IFJ.
Pokusné odevzdání: úterý 18. listopadu 2025, 23:59 (nepovinné).
Datum odevzdání: středa 3. prosince 2025, 23:59.
Způsob odevzdání: prostřednictvím StudIS, aktivita „Projekt – Registrace a Odevzdání“.

Hodnocení:

- Do předmětu IFJ získá každý maximálně 28 bodů: 18 celková funkčnost projektu (tzv. programová část), 5 dokumentace, 5 obhajoba.
- Do předmětu IAL získá každý maximálně 15 bodů: 5 celková funkčnost projektu, 5 obhajoba, 5 dokumentace.
- Lze získat také až 5 bonusových bodů do předmětu IFJ, a to za implementaci bonusových rozšíření, tvůrčí přístup, zajímavé řešení nebo aktivitu na fóru. Nelze však získat více než 35 % bodů z vašeho individuálního hodnocení základní funkčnosti.

- **Udělení zápočtu z IFJ i IAL je podmíněno získáním min. 20 bodů v průběhu semestru. Navíc v IFJ z těchto 20 bodů musíte získat nejméně 4 body za programovou část projektu.**
- Dokumentace bude hodnocena nejvýše polovinou bodů z hodnocení funkčnosti projektu, bude také reflektovat procentuální rozdělení bodů a bude zaokrouhlena na celé body.
- Body zapisované za programovou část včetně rozšíření budou také zaokrouhleny a v případě přesáhnutí 18 bodů zapsány do termínu „Projekt – Bonusové hodnocení“ v IFJ.

Řešitelské týmy:

- Projekt budou řešit čtyřčlenné týmy. Týmy s jiným počtem členů jsou nepřípustné (jen výjimečně tříčlenné).
- Vytváření týmů se provádí funkcionalitou Týmy v IS VUT. Tým vytváří vedoucí a název týmu bude automaticky generován na základě loginu vedoucího. Vedoucí má kontrolu nad složením týmu, smí předat vedení týmu jinému členovi a bude odevzdávat výsledný archiv. Rovněž vzájemná komunikace mezi vyučujícími a týmy bude probíhat nejlépe prostřednictvím vedoucích (ideálně v kopii dalším členům týmu).
- Jakmile bude mít tým dostatek členů, bude mu umožněno si zaregistrovat jednu ze dvou variant zadání (viz sekci 7) v aktivitě „Projekt – Registrace a odevzdání“.
- Všechny hodnocené aktivity k projektu najdete ve StudIS, studijní informace v Moodle předmětu IFJ a další informace na stránkách předmětu[1].

2 Zadání

Vytvořte program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ25 a přeloží jej do cílového jazyka IFJcode25 (tzv. mezikód). Jestliže proběhne překlad bez chyb, vrací se návratová hodnota 0 (nula). Pokud došlo k nějaké chybě překladu, vrací se návratová hodnota následovně:

1	chyba v programu v rámci lexikální analýzy (chybná struktura aktuálního lexému).
2	chyba v programu v rámci syntaktické analýzy (chybná syntaxe programu, chybějící kostra atp.).
3	sémantická chyba – použití nedefinované funkce či proměnné.
4	sémantická chyba – redefinice funkce či proměnné.
5	statická sémantická chyba – neočekávaný počet argumentů při volání funkce nebo špatný typ parametru vestavěné funkce.
6	statická sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.
10	ostatní sémantické chyby.
99	interní chyba překladače (tj. neovlivněná vstupním programem, např. chyba alokace paměti.

Vzhledem k dynamické povaze jazyka bude nutné některé sémantické typové kontroly provádět až při běhu (překladač bude generovat kód, který tyto kontroly provede). Pokud je při běhu odhalena taková chyba, program se ukončí (příslušnou instrukcí) s následujícím návratovým kódem:

25	běhová sémantická chyba – špatný typ parametru vestavěné funkce.
26	běhová sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.

[1] http://www.fit.vut.cz/study/courses/IFJ/public/project

Překladač bude načítat řídicí program v jazyce IFJ25 ze standardního vstupu a generovat výsledný kód v jazyce IFJcode25 (viz sekci 10) na standardní výstup. Všechna chybová hlášení, varování a ladicí výpisy provádějte na standardní chybový výstup; tj. bude se jednat o konzolovou aplikaci bez grafického uživatelského rozhraní (tzv. filtr). Pro interpretaci výsledného programu v cílovém jazyce IFJcode25 bude v Moodle předmětu k dispozici interpret.

Klíčová slova jsou sázena tučně a některé lexémy jsou pro zvýšení čitelnosti v apostrofech, přičemž znak apostrofu není v takovém případě součástí jazyka!

3 Popis jazyka, lexikální jednotky, datové typy

Jazyk IFJ25 je zjednodušenou podmnožinou jazyka Wren[2], což je jednoduchý, skriptovací, dynamicky typovaný a objektově orientovaný jazyk, jehož hlavním způsobem užití je integrace do jiných aplikací (např. do herních enginů).

V programovacím jazyce IFJ25 **záleží** na velikosti písmen u identifikátorů i klíčových slov (tzv. *case-sensitive*). Jazyk je dynamicky typovaný.

- *Identifikátor* (lokální proměnné, uživatelské funkce, getteru) je neprázdná posloupnost číslic, písmen (malých i velkých) a znaku podtržítka ('_') začínající písmenem. Jazyk IFJ25 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam[3], a proto se nesmí vyskytovat jako identifikátory:

  Klíčová slova: **`class`**, **`if`**, **`else`**, **`is`**, **`null`**, **`return`**, **`var`**, **`while`**, **`Ifj`**, **`static`**, **`true`**, **`false`**, **`Num`**, **`String`**, **`Null`**

- *Identifikátor vestavěné funkce IFJ25* je definován jako tečkou oddělený jmenný prostor '**`Ifj`**' a identifikátor konkrétní funkce (viz sekci 6), pro který platí stejná pravidla jako pro *identifikátor*. Mezi jednotlivými částmi se může vyskytovat libovolný počet bílých znaků. Příklad: **`Ifj . write`**.
- *Identifikátor globální proměnné* je neprázdná posloupnost číslic, písmen a znaku podtržítka, která začíná dvěma znaky podtržítka ('__'), za nimiž následují libovolné povolené znaky.
- *Celočíselný literál* (rozsah int v IFJcode25) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě. Podporován je také hexadecimální literál, který začíná sekvencí **`0x`**, za kterou následují hexadecimální číslice (**`0`**–**`9`**, **`a`**–**`f`**, **`A`**–**`F`**).
- *Desetinný literál* (rozsah float v IFJcode25) také vyjadřuje nezáporná čísla v desítkové soustavě, přičemž literál je tvořen celou a desetinnou částí, nebo celou částí a exponentem, nebo celou a desetinnou částí a exponentem. Celá i desetinná část je tvořena neprázdnou posloupností číslic. Exponent je celočíselný, začíná znakem '**`e`**' nebo '**`E`**', následuje nepovinné znaménko '**`+`**' (plus) nebo '**`-`**' (mínus) a poslední částí je neprázdná posloupnost číslic. Mezi jednotlivými částmi nesmí být jiný znak, celou a desetinnou část odděluje znak '`.`' (tečka)[4].
- *Řetězcový literál* je oboustranně ohraničen dvojitými uvozovkami (`"`, ASCII hodnota 34). Tvoří jej libovolný počet znaků zapsaných na jednom řádku (nemůže obsahovat odřádkování). Možný je i prázdný řetězec (`""`). Znaky s ASCII hodnotou větší než 31 (mimo

[2] Online dokumentace jazyka Wren: `https://wren.io/`. Viz také sekci 9.

[3] V rozšířeních mohou být použita i další klíčová slova, která ale budeme testovat pouze v případě implementace patřičného rozšíření.

[4] Přebytečné počáteční číslice `0` jsou zakázány v celočíselné části, ale nevadí v exponentu, kde jsou ignorovány.

`"`) lze zapisovat přímo. Některé další znaky lze zapisovat pomocí escape sekvence: '`\"`', '`\n`', '`\r`', '`\t`', '`\\`'. Jejich význam se shoduje s odpovídajícími escape sekvencemi jazyka Wren. Pokud znaky za zpětným lomítkem neodpovídají žádnému z uvedených vzorů, dojde k chybě 1.
Znak v řetězci může být zadán také pomocí escape sekvence '`\x`*dd*', kde *dd* je hexadecimální číslo složeno z právě dvou hexadecimálních číslic[5].
Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

```
"Ahoj\n\"Sve'te \\\x22"
```

reprezentuje řetězec

```
Ahoj
"Sve'te  \"
```

. Neuvažujte řetězce, které obsahují vícebajtové znaky kódování Unicode (např. UTF-8).

- *Víceřádkový řetězcový literál* je oboustranně ohraničen trojitými uvozovkami (`"""`). Tvoří jej libovolný počet znaků včetně znaku odřádkování. Literál obsahuje všechny znaky mezi trojitými uvozovkami, neplatí zde žádné escape sekvence. Jedinou výjimkou je případ, kdy se na řádku za počátečními nebo před ukončujícími trojitými uvozovkami nachází *pouze* bílé znaky (mezery, tabulátory, odřádkování) – tyto bílé znaky pak nejsou součástí literálu (v případě ukončujících uvozovek navíc pak není součástí literálu ani poslední znak nového řádku). Například v kódu

  ```
  var x = """A
    hoj s\x22"vete
     """
  ```

  je hodnota `x` ekvivalentní s řetězcovým literálem `"A\n  hoj s\\x22\"vete"`.
- *Datové typy* jsou `Num`, `String` a `Null`. Jazyk obsahuje pouze jeden typ pro celá i desetinná čísla. Hodnotou proměnné i návratovou hodnotou funkce může být speciální hodnota `null`, což je jediná hodnota typu `Null`.
- *Term* je libovolný literál (celočíselný, desetinný, řetězcový či `null`) nebo identifikátor proměnné, globální proměnné či statického getteru.
- *Řádkový komentář* začíná dvojicí lomítek ('`//`', ASCII hodnota 47) a za komentář je považováno vše, co následuje až do konce řádku. Výskyt blokového komentáře se pro účely syntaktické analýzy považuje za jeden znak nového řádku.
- *Blokový komentář* začíná posloupností symbolů '`/*`' a je ukončen dvojicí symbolů '`*/`'. Podporovány jsou i vnořené blokové komentáře! Výskyt blokového komentáře se pro účely syntaktické analýzy považuje za jeden bílý znak.

4 Struktura jazyka

IFJ25 je strukturovaný programovací jazyk podporující definice modifikovatelných proměnných (lokálních i globálních) a uživatelských funkcí včetně jejich rekurzivního volání. Podporovány jsou také speciální typy funkcí „zastupujících proměnné“, tzv. statické gettery a settery, ke kterým se syntakticky přistupuje jako k proměnným, ale sémanticky dochází k provedení těla funkce.

4.1 Základní struktura programu

Program se skládá z povinného prologu a povinné kostry, uvnitř které se nachází sekvence definic uživatelských funkcí včetně definice *hlavní funkce* `main`, která je povinným vstupním bodem

[5] Na rozdíl od jazyka Wren není nutné podporovat sekvence začínající '`\u`'.

programu. Chybějící funkce **`main()`** (bez parametrů[6]) způsobí chybu 3.

Prolog se skládá z jednoho řádku:

```
import "ifj25" for Ifj
```

Prolog mohou libovolně prokládat bílé znaky (za řetězcem **`"ifj25"`** však není dovoleno odřádkování). Slouží především pro zachování kompatibility s jazykem Wren. Jeho absence povede na syntaktickou chybu 2.

Za prologem následuje povinně kostra programu:

```
class Program {
    // Sekvence funkci
}
```

Poznámka: Wren je třídní objektově orientovaný jazyk, ve kterém hrají třídy (classes) podstatnou roli. To, co zde označujeme jako funkce, jsou v jazyce Wren ve skutečnosti tzv. statické metody, a naše globální proměnné jsou tzv. statické (třídní) atributy. V předmětu IFJ však aspekty objektové orientace zatím neřešíme – jazyk IFJ25 považujeme za strukturovaný. Na povinnou kostru zde tedy nahlížejte pouze jako na syntaktický požadavek.

Poznámka: V příkladech níže jsou pro úsporu místa prolog a někdy i kostra vynechány.

Uvnitř kostry programu se nachází definice funkcí (vč. getterů a setterů). V tělech funkcí lze potom používat *příkazy* (viz dále). Mimo kostru programu se nachází **pouze** prolog.

Před, za i mezi jednotlivými tokeny se může vyskytovat libovolný počet bílých znaků mimo znak nového řádku (mezera, tabulátor, víceřádkový komentář). Znak nového řádku (dále označen taky jako ⟨*EOL*⟩) v jazyce IFJ25 slouží k oddělování příkazů, není tedy možné na jednom řádku zapsat více příkazů. Znak nového řádku je (kromě případů, kde je povinný) možné použít *za* otevíracími závorkami, tečkami, čárkami, operátory a znakem '**=**'. Sekvence několik znaků nového řádku se považuje za jeden znak nového řádku.

```
// Lze:
Ifj .   write( "Ahoj svete")
Ifj.
  write("Ahoj svete")
static func1(
    x) {
     Ifj.write("Ahoj")
}
// Nelze:
Ifj.write("Ahoj svete")  Ifj.write("Ahoj svete")
static func2(x) {
    Ifj.write("Ahoj") } // prikaz volani funkce neni zakoncen novym radkem
```

Struktura definice uživatelských funkcí a jednotlivé příkazy jsou popsány v následujících sekcích.

4.2 Bloky

Blokem se rozumí sekvence příkazů (příkazy jsou definovány níže v sekci 4.6) uzavřená ve složených závorkách, přičemž za otevírací závorkou je povinně znak nového řádku:

{ ⟨*EOL*⟩
sekvence_příkazů_a_bloků
}

[6]Pokud je bezparametrická varianta funkce **`main`** definována, je možné navíc definovat i přetíženou verzi s parametry (viz sekci 4.4).

Blok může být i prázdný (bez příkazů), i tehdy však musí za otvírací závorkou být znak nového řádku. Vizte také dobrovolné rozšíření ONELINEBLOCK, které přidává podporu pro speciální jednořádkové bloky.

Blok se využívá jako tělo funkce, větve podmíněného příkazu (**`if`**) nebo cyklu (**`while`**). Bloky se mohou vnořovat (na místě příkazu je nový blok), přičemž vždy tvoří nový *rozsah platnosti* (scope) pro lokální proměnné.

4.3 Proměnné

Proměnné jazyka IFJ25 jsou lokální a globální:

Globální proměnné začínají dvěma podtržítky, jsou přístupné v celém programu a před použitím je není nutné definovat. Výsledkem čtení nedefinované globální proměnné je hodnota **`null`**.

Lokální proměnné začínají písmenem, jsou definované v uživatelských funkcích a jejich podblocích. Lokální proměnné mají rozsah platnosti od místa jejich definice až po konec bloku, ve kterém byly definovány. Před použitím je nutné je definovat příkazem **`var`**. Přístup k nedefinované lokální proměnné vede na chybu 3.

Proměnné jsou dynamicky typované, jejich typ se tak může při přiřazení změnit.

V podblocích lze překrýt (angl. *shadowing*) proměnnou definovanou v nadbloku. Uvnitř jednoho bloku však není možné dvakrát definovat proměnnou stejného jména (vede na chybu 4). Globální proměnnou není možné překrýt (lokální proměnné mají jiná pravidla pro identifikátory).

Příklad:

```
var a
a = 123
a = "outer" // dynamicka zmena typu promenne
{
    var a
    a = "inner"
    Ifj.write(a) // vypise "inner"
    // Nyni ale nelze znovu (redefinice):
    // var a
}
Ifj.write(a) // vypise outer
```

4.4 Deklarace a definice uživatelských funkcí

Definice funkce se skládá z hlavičky a těla funkce. Definice funkce je zároveň její deklarací. Každá použitá funkce musí být definovaná, jinak končí analýza chybou 3. V IFJ25 nelze definovat vnořené funkce (šlo by o syntaktickou chybu 2).

Definice funkce nemusí lexikálně předcházet kódu pro použití této funkce, tzv. *volání funkce* (příkaz volání je definován v sekci 4.6). Je tak možné například zapsat vzájemné rekurzivní volání funkcí (tj. funkce `foo` volá funkci `bar`, která opět může volat funkci `foo`).

Příklad vzájemné rekurze (bez ukončující podmínky):

```
class Program {
    static main() {
        foo(10)
    }
    static bar(param) {
        return foo(param)
    }
```

```
    static foo(param) {
        var next
        next = bar(param)
        return next - 1
    }
}
```

Přetěžování

V IFJ25 je povoleno přetěžování funkcí (angl. *overloading*) podle arity – je možné definovat dvě funkce stejného jména, ale pouze tehdy, když mají rozdílný počet parametrů:

```
static foo(par1, par2) {
    return par1 + par2
}
static foo(only_par) {
    return foo(only_par, only_par)
}
// Nyni uz nelze:
static foo(par) {
}
```

Syntaxe definice funkce

Definice funkce je konstrukce (hlavička a tělo) ve tvaru:

`static` *id* **`(`** *seznam_parametrů* **`)`** *blok* ⟨*EOL*⟩

Hlavička definice funkce sahá od klíčového slova **`static`** až po pravou kulatou závorku, pak následuje tělo funkce tvořené blokem (viz sekci 4.2) a znakem nového řádku. *Seznam_parametrů* je tvořen posloupností identifikátorů oddělených čárkou, přičemž za posledním parametrem se čárka nesmí uvádět. Seznam může být i prázdný.

Parametry

Identifikátor parametru slouží jako identifikátor v těle funkce pro získání hodnoty tohoto parametru. Parametry jsou vždy předávány hodnotou. Uvnitř těla funkce se parametry chovají jako předdefinované (ale jinak běžné) lokální proměnné s předanou hodnotou.

Návratová hodnota

Výsledek funkce je dán hodnotou provedeného příkazu návratu z funkce (viz sekci 4.6). Po provedení příkazu návratu z funkce je funkce ukončena. Pokud funkce neobsahuje příkaz návratu z funkce, je ukončena po naposledy provedeném příkazu svého těla a vrací hodnotu **`null`**.

4.5 Funkce zastupující proměnnou (statický getter a setter)

Funkce typu statický getter je speciální, syntakticky odlišenou podobou uživatelské funkce, která nemá žádný parametr a při jejím volání se nepoužívají kulaté závorky. Funkce typu statický setter je speciální typ funkce, která má právě jeden parametr a pro její volání se používá příkaz přiřazení. Použití funkcí zastupujících proměnnou tedy připomíná práci s proměnnou, sémanticky se však chovají stejně jako běžné funkce a mohou provádět libovolnou sekvenci příkazů. Statický getter je možné použít na místě termu. V souborech k projektu (Moodle) najdete několik příkladů, které tyto speciální funkce demonstrují.

V kódu může existovat nejvýše jeden statický getter a jeden statický setter se stejným identifikátorem (nemusí existovat oba). Zároveň může existovat (na nich zcela nezávislá) standardní uživatelská funkce se stejným identifikátorem (a libovolným počtem parametrů).

Definice statického getteru je konstrukce ve tvaru:

`static` *id blok* ⟨*EOL*⟩

Definice statického setteru je konstrukce ve tvaru:

`static` *id* **`=(`** *id* **`)`** *blok* ⟨*EOL*⟩

4.6 Syntaxe a sémantika příkazů

Následuje seznam různých typů *příkazů*, které lze využít v rámci sekvence příkazů uvnitř bloku (viz sekci 4.2). Nejprve je uvedena syntaxe příkazu, pak případné poznámky k syntaxi a poté sémantika příkazu. Za každým příkazem se očekává znak nového řádku ⟨*EOL*⟩!

Příkaz definice lokální proměnné:

`var` *idl*

Levý operand *idl* je identifikátor lokální proměnné.

Příkaz definuje lokální proměnnou, která v aktuálním bloku ještě nebyla definována (jinak dojde k chybě 4). Hodnota proměnné je nastavena na **`null`**.

Vizte také dobrovolné rozšíření EXTSTAT.

Příkaz přiřazení:

id **`=`** *výraz*

Levý operand *id* je identifikátor lokální proměnné nebo identifikátor globální proměnné nebo identifikátor statického setteru.

Příkaz provádí vyhodnocení výrazu *výraz* (viz sekci 5) a přiřazení jeho hodnoty do levého operandu *id*, který je lokální proměnnou (tj. dříve definovanou pomocí klíčového slova **`var`**) nebo globální proměnnou (předem se nedefinuje, vzniká po prvním přiřazení).

V případě, že je levým operandem identifikátor statického setteru, sémantika odpovídá zavolání příslušné funkce, přičemž je hodnota vyhodnoceného výrazu předána jako jediný parametr.

Podmíněný příkaz:

`if (`*výraz***`)`** $blok_1$ **`else`** $blok_2$

V základním zadání je vždy přítomná i část **`else`**. Za ukončovací kulatou závorkou, za koncem $bloku_1$ ani za klíčovým slovem **`else`** nesmí být znak nového řádku.

Nejprve se vyhodnotí *výraz*, který je libovolného typu. Pokud je vyhodnocený výraz pravdivý, vykoná se sekvence příkazů uvnitř $bloku_1$, jinak se vykoná sekvence příkazů uvnitř $bloku_2$.

Pravdivost výrazu se posuzuje takto: pokud je výsledná hodnota výrazu přímo pravdivostní (tj. pravda či nepravda – v základním zadání pouze jako výsledek aplikace relačních operátorů dle sekce 5), je použita tak, jak je. Pokud je výsledná hodnota **`null`**, výraz je nepravdivý. Ve všech ostatních případech je výraz pravdivý.

Vizte také dobrovolné rozšíření BOOLTHEN.

Příkaz cyklu:

`while (`*výraz***`)`** *blok*

Znak nového řádku nesmí být mezi ukončovací kulatou závorkou a začátkem *bloku*.

Opakuje provádění sekvence příkazů uvnitř *bloku* tak dlouho, dokud je hodnota *výrazu* pravdivá. Pravidla pro vyhodnocování pravdivosti výrazu jsou stejná jako u podmíněného příkazu.

Vizte také dobrovolné rozšíření CYCLES.

Volání vestavěné či uživatelem definované funkce nebo funkce typu getter:

id **=** *id_funkce* **(***seznam_vstupních_parametrů***)** nebo
id **=** *id_fce_getter*

Levý operand *id* je identifikátor lokální proměnné nebo identifikátor globální proměnné nebo identifikátor statického setteru.

Seznam_vstupních_parametrů je seznam termů oddělených čárkami, který nesmí končit čárkou. Seznam vstupních parametrů může být i prázdný. Parametrem volání funkce není v základním zadání výraz! Podporu výrazů je možné doplnit v rámci dobrovolného rozšíření FUNEXP.

V případě, že příkaz volání funkce obsahuje jiný počet skutečných parametrů, než funkce očekává (tedy než je uvedeno v její hlavičce, a to i u vestavěných funkcí), jedná se o chybu 5.

Sémantika volání vestavěných funkcí je popsána pro jednotlivé funkce v sekci 6.

Sémantika volání uživatelsky definovaných funkcí (vč. statických getterů) je následující: Příkaz zajistí předání parametrů hodnotou a předání řízení do těla funkce. Po dokončení provádění zavolané funkce je přiřazena návratová hodnota do proměnné *id* (příp. *idl*) a běh programu pokračuje bezprostředně za příkazem volání právě provedené funkce.

Vizte také dobrovolné rozšíření EXTSTAT.

Příkaz návratu z funkce:

`return` *výraz*

Příkaz může být použit v těle libovolné funkce (vč. **`main`**). Nejprve se vyhodnotí povinný *výraz*, čímž se získá návratová hodnota. Poté dojde k ukončení provádění těla funkce a návratu do místa volání, kam funkce vrátí vypočtenou návratovou hodnotu.

Vizte také dobrovolné rozšíření EXTSTAT.

Samostatný výraz na místě příkazu

Volitelně můžete podporovat na místě příkazu i samostatně (na řádku) stojící *výraz*. Jeho hodnota bude vyhodnocena, ale zahozena. Absence implementace tohoto typu příkazu nebude považována za chybu, tj. nebudeme to v základu testovat. Tato vlastnost se vám může hodit zejména při některých způsobech implementace rozšíření FUNEXP.

5 Výrazy

Výrazy jsou tvořeny termy, závorkami, aritmetickými, řetězcovými a relačními operátory a operátory testu typu. V základním zadání není možné uvnitř výrazu volat běžné funkce (ani bezparametrické), pouze funkce zastupující proměnnou (statické gettery).

Vzhledem k dynamické povaze jazyka není možné provést úplnou typovou kontrolu při překladu, očekává se však typová kontrola minimálně ve výrazech s literály (např. výraz `"1" + 2` skončí chybou překladu 6). Za realizaci podrobnější statické analýzy typů je možné získat bonusové body (viz dobrovolné rozšíření STATICAN). Ostatní typové kontroly je nutné provádět při běhu programu (překladač generuje kód, který typové kontroly provádí). V případě detekce typové neshody pak interpretace končí chybou 26. Jazyk IFJ25 neobsahuje žádné implicitní typové konverze (řetězce a čísla je nutné mezi sebou explicitně převádět použitím vestavěných funkcí).

Řetězcové operátory

Mezi dvěma výrazy typu **`String`** je možné použít binární *konkatenační* operátor **`+`**, který značí konkatenaci řetězců.

Iterační operátor **`*`** je možné použít mezi výrazem *s* typu **`String` nalevo** a výrazem *n* typu **`Num` napravo**. Operátor značí iteraci řetězce, tedy výsledkem je *n*-krát za sebou zkonkatenovaný řetězec *s* (příklad: `("a" * 3) == "aaa"`). Při použití iteračního operátoru musí být pravý operand *n* celočíselný, jinak dojde k chybě 6 (překlad) / 26 (interpretace).

Aritmetické operátory

Standardní binární operátory nad dvěma výrazy typu **`Num`** jsou **`+`**, **`-`**, **`*`**, **`/`** a značí sčítání, odčítání[7], násobení a dělení. Jediný číselný typ **`Num`** odpovídá typu *float* z cílového jazyka IFJcode25, všechny číselné operace jsou tedy v plovoucí řádové čárce.

Je-li jeden z operandů typu **`Num`** a jeden typu **`String`**, dojde k chybě 6 / 26. Výjimkou je pouze iterační operátor **`*`** (viz výše).

Relační (porovnávací) operátory

Relační operátory jsou **`<`**, **`>`**, **`<=`**, **`>=`**, **`==`** a **`!=`**. Výsledkem porovnání je pravdivostní hodnota. Jejich sémantika je standardní, odpovídá chování v jazyce Wren.

Pro operátory **`==`** a **`!=`** platí, že operandem na kterékoliv straně je výraz libovolného typu nebo speciální hodnota **`null`**. Pokud jsou operandy stejného typu, tak se porovnají jejich hodnoty. Pokud jsou operandy jiného typu, výsledkem je vždy nepravda. Výraz **`null == null`** je pravdivý. Operátor **`!=`** je negací operátoru **`==`**.

Pro ostatní relační operátory platí, že operandy na obou stranách jsou výrazy typu **`Num`**. Pokud nějaký z operandů není tohoto typu, dojde k chybě 6 / 26.

Bez rozšíření BOOLTHEN není s výsledkem porovnání (pravdivostní hodnota) možné dále pracovat (např. uložit jej do proměnné), lze jej tedy využít pouze jako podmínku příkazů **`if`** a **`while`**.

[7] Číselné literály jsou v základním zadání sice nezáporné, ale výsledek výrazu přiřazený do proměnné již záporný být může.

Operátor testu typu

Operátorem testu typu je `is`. Levým operandem je výraz, pravým operátorem je pouze klíčové slovo označující typ – `String`, `Num` nebo `Null`. Výsledkem je pravdivostní hodnota, která udává, zda je výraz na levé straně typu určeného pravou stranou.

5.1 Priorita operátorů

Prioritu operátorů lze explicitně upravit závorkováním podvýrazů. Následující tabulka udává priority operátorů (nahoře nejvyšší):

Priorita	Operátory	Asociativita
1	`*` `/`	levá
2	`+` `-`	levá
3	`<` `>` `<=` `>=`	levá
4	`is`	levá
5	`==` `!=`	levá

6 Vestavěné funkce

Překladač bude poskytovat několik základních vestavěných funkcí, které bude možné využít v programech v jazyce IFJ25. Pro generování kódu vestavěných funkcí je vhodné (a obvykle nutné) využít specializovaných instrukcí jazyka IFJcode25. Všechny vestavěné funkce jsou zařazeny do jmenného prostoru `Ifj`.

Podobně jako u výrazů se očekává kontrola typů parametrů minimálně u literálů, kdy při použití špatného typu literálu jako parametru následujících vestavěných funkcí dojde k chybě překladu 5. Je však nutné generovat kód, který typy kontroluje za běhu a v případě nekompatibilního typu parametru ukončí interpretaci s kódem 25.

Ve textu níže je typ návratové hodnoty funkce označen za šipkou, typ parametru za dvojtečkou – jde jen o značení pro účely dokumentace, ne o syntakticky správnou konstrukci jazyka. Symbol `|` znamená sjednocení typů (spojka „nebo“).

Vestavěné funkce pro načítání literálů a výpis termů:

- *Funkce pro načítání hodnot ze vstupu*:
  `static Ifj.read_str()` → `String | Null`
  `static Ifj.read_num()` → `Num | Null`
  Vestavěné funkce ze standardního vstupu načtou jeden řádek ukončený odřádkováním nebo koncem souboru (EOF). Funkce `read_str` tento řetězec vrátí bez symbolu konce řádku. Načítaný řetězec nepodporuje escape sekvence! Pokud je načten pouze EOF, vrací `null`. Funkce `read_num` načítá celé nebo desetinné číslo. Pokud je na vstupu načten jakýkoli nevhodný znak (včetně okolních bílých), vrací `null`. V případě chybějící hodnoty na vstupu (např. pouze načtení EOF) vrací `null`.
- *Funkce pro výpis hodnoty*:
  `static Ifj.write(`*term*`)` → `Null`
  Vypíše hodnotu termu *term* (libovolného typu) na standardní výstup ihned a bez žádných oddělovačů dle typu v patřičném formátu: Termy typu `String` vypíše tak, jak jsou. Termy typu `Num` s celočíselnou hodnotou (tj. s nulovou desetinnou částí) vypíše jako celé číslo

obvyklým způsobem (odpovídá specifikátoru `'%d'`[8]), desetinné hodnoty termu typu **`Num`** vypíše ve formátu, který odpovídá specifikátoru `'%a'`[9]. Hodnota **`null`** je tištěna jako literál **`"null"`**. Tyto požadavky odpovídají chování instrukce **WRITE** v IFJcode25 pro symboly typu string, int (!), float a nil.

Vestavěné funkce pro konverzi typů:

- **`static Ifj.floor(`***term* **`: Num)`** → **`Num`**
  Vrátí hodnotu desetinného termu *term* převedenou na celé číslo oříznutím desetinné části. Pro konverzi z desetinného čísla využijte odpovídající instrukci/-e z IFJcode25.
- **`static Ifj.str(`***term***`)`** → **`String`**
  Vrátí řetězcovou reprezentaci termu *term* (libovolného typu). Do řetězců jsou hodnoty tisknuty obdobně jako ve funkci **`Ifj.write`** – s rozdílem, že desetinné hodnoty s nenulovou desetinnou částí tiskne ve formátu odpovídajícím specifikátoru `'%.2f'` (v souladu s instrukcí **FLOAT2STR**).

Vestavěné funkce pro práci s řetězci:

- **`static Ifj.length(`***s* **`: String)`** → **`Num`**
  Vrátí délku (počet znaků) řetězce *s*.
- **`static Ifj.substring(`***s* **`: String,`** *i* **`: Num,`** *j* **`: Num)`** → **`String | Null`**
  Vrátí podřetězec zadaného řetězce *s*. Druhým parametrem *i* je dán index začátku požadovaného podřetězce (počítáno od nuly) a třetím parametrem *j* určuje index za posledním znakem podřetězce (též počítáno od nuly).
  Funkce dále vrací hodnotu **`null`**, nastane-li některý z těchto případů:
  - $i < 0$
  - $j < 0$
  - $i > j$
  - $i \geq$ **`Ifj.length`**(s)
  - $j >$ **`Ifj.length`**(s)

  Překlad / interpretace končí chybou 6 / 26, pokud parametr *i* nebo *j* není celočíselný.
- **`static Ifj.strcmp(`***s1* **`: String,`** *s2* **`: String)`** → **`Num`**
  Analogicky jako stejnojmenná funkce standardní knihovny jazyka C vrací výsledek lexikografického porovnání dvou řetězců (-1 pro *s1* menší než *s2*, 0 pro rovnost, 1 pro *s1* větší než *s2*).
- **`static Ifj.ord(`***s* **`: String,`** *i* **`: Num)`** → **`Num`**
  Vrátí ordinální hodnotu (ASCII) *i*-tého znaku v řetězci *s* (indexováno od nuly). Je-li řetězec prázdný nebo index mimo jeho meze, vrací funkce hodnotu **`0`**.
  Překlad / interpretace končí chybou 6 / 26, pokud parametr *i* není celočíselný.
- **`static Ifj.chr(`***i* **`: Num)`** → **`String`**
  Vrátí jednoznakový řetězec se znakem, jehož ASCII kód je zadán parametrem *i*. Hodnotu *i* mimo interval $\langle 0; 255 \rangle$ řeší odpovídající instrukce IFJcode25.
  Překlad / interpretace končí chybou 6 / 26, pokud parametr *i* nebo *j* není celočíselný.

7 Implementace tabulky symbolů

Tabulka symbolů bude implementována pomocí abstraktní datové struktury, která je ve variantě zadání pro daný tým označena identifikátory BVS a TRP, a to následovně:

[8] Specifikátor formátu v rodině standardní funkce `printf` jazyka C (standard C99 a novější).
[9] Specifikátor formátu funkce `printf` pro přesnou hexadecimální reprezentaci desetinného čísla.

vv-BVS) Tabulku symbolů implementujte pomocí **výškově vyváženého** binárního vyhledávacího stromu.

TRP-izp) Tabulku symbolů implementujte pomocí tabulky s rozptýlenými položkami s **implicitním zřetězením položek** (TRP s otevřenou adresací).

Implementace tabulky symbolů bude uložena v souboru `symtable.c` (případně `symtable.h`). Další požadavky jsou uvedeny v sekci 12.2.

8 Příklady

Níže je uveden jednoduchý příklad programu v jazyce IFJ25. Další příklady najdete v Moodle IFJ v sekci s projektem. V souboru `ex0-vsechny-konstrukce.wren` je příklad kódu, ve kterém je užita většina syntaktických prostředků jazyka.

Výpočet faktoriálu (rekurzivně)

```
// Program 2: Vypocet faktorialu (rekurzivne)
import "ifj25" for Ifj
class Program {
    // Hlavni funkce
    static main() {
        Ifj.write("Zadejte cislo pro vypocet faktorialu: ")
        var inp
        inp = Ifj.read_num()
        if (inp != null) {
            if (inp < 0) {
                Ifj.write("Faktorial nelze spocitat!\n")
            } else {
                // Overime celociselnost
                var inpFloored
                inpFloored = Ifj.floor(inp)
                if (inp == inpFloored) {
                    var vysl
                    vysl = factorial(inp)
                    vysl = Ifj.str(vysl)
                    Ifj.write("Vysledek: ")
                    Ifj.write(vysl)
                } else {
                    Ifj.write("Cislo neni cele!\n")
                }
            }
        } else {
            Ifj.write("Chyba pri nacitani celeho cisla!\n")
        }
    }
    // Funkce pro vypocet hodnoty faktorialu
    static factorial(n) {
        var result
        if (n < 2) {
            result = 1
        } else {
            var decremented_n
            decremented_n = n - 1
            result = factorial(decremented_n)
            result = n * result
        }
        return result
```

```
    }
}
```

9 Doporučení k testování

Programovací jazyk IFJ25 je schválně navržen tak, aby byl (téměř) kompatibilní s podmnožinou jazyka Wren[10]. Pokud si nejste jistí, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ25, můžete si to ověřit pomocí interpretu `wren-cli` na serveru Merlin. V interpretu je již vestavěn modul `ifj25` s definicemi vestavěných funkcí jazyka IFJ25. Stačí tedy váš program v jazyce IFJ25 uložit například do souboru `testPrg.wren` a následně jej spustit příkazem:

```
/pub/courses/ifj/ifj25/wren-cli testPrg.wren < test.in > test.out
```

Tím lze jednoduše zkontrolovat, co by měl provést zadaný zdrojový kód (resp. co by při interpretaci měl vykonávat vygenerovaný cílový kód). Je ale potřeba si uvědomit, že jazyk Wren je nadmnožinou jazyka IFJ25, a tudíž může zpracovat i konstrukce, které nejsou v IFJ25 vyžadované či povolené (např. bohatší syntaxe a sémantika většiny příkazů, či dokonce zpětné nekompatibility). Některé známé odlišnosti budou uvedeny v Moodle IFJ a můžete je diskutovat tamtéž ve fóru k projektu. Typicky platí, že oceňujeme podporu různých funkcí a konstrukcí jazyka Wren, které jsou nad rámec zadání IFJ25, ale je vhodné to radši konzultovat na fóru.

10 Cílový jazyk IFJcode25

Cílový jazyk IFJcode25 je mezikódem, který zahrnuje instrukce tříadresné (typicky se třemi argumenty) a zásobníkové (typicky bez parametrů a pracující s hodnotami na datovém zásobníku). Každá instrukce se skládá z operačního kódu (klíčové slovo s názvem instrukce), u kterého nezáleží na velikosti písmen (tj. case insensitive). Zbytek instrukcí tvoří operandy, u kterých na velikosti písmen záleží (tzv. case sensitive). Operandy oddělujeme libovolným nenulovým počtem mezer či tabulátorů. Odřádkování slouží pro oddělení jednotlivých instrukcí, takže na každém řádku je maximálně jedna instrukce a není povoleno jednu instrukci zapisovat na více řádků. Každý operand je tvořen proměnnou, konstantou nebo návěštím. V IFJcode25 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode25 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode25
```

10.1 Hodnoticí interpret `ic25int`

Pro hodnocení a testování mezikódu v IFJcode25 je k dispozici interpret pro příkazovou řádku (`ic25int`):

```
ic25int prg.code < prg.in > prg.out
```

Chování interpretu lze upravovat pomocí přepínačů/parametrů příkazové řádky. Nápovědu k nim získáte pomocí přepínače `--help`.

Proběhne-li interpretace bez chyb, vrací se návratová hodnota `0` (nula). Chybovým případům odpovídají následující návratové hodnoty:

- 50 - chybně zadané vstupní parametry na příkazovém řádku při spouštění interpretu.
- 51 - chyba při analýze (lexikální, syntaktická) vstupního kódu v IFJcode25.

[10] Online dokumentace k Zig: https://wren.io/

- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode25.
- 53 - běhová chyba interpretace – špatné typy operandů.
- 54 - běhová chyba interpretace – přístup k neexistující proměnné (rámec existuje).
- 55 - běhová chyba interpretace – rámec neexistuje (např. čtení z prázdného zásobníku rámců).
- 56 - běhová chyba interpretace – chybějící hodnota (v proměnné, na datovém zásobníku, nebo v zásobníku volání).
- 57 - běhová chyba interpretace – špatná hodnota operandu (např. dělení nulou, špatná návratová hodnota instrukce EXIT).
- 58 - běhová chyba interpretace – chybná práce s řetězcem.
- 60 - interní chyba interpretu tj. neovlivněná vstupním programem (např. chyba alokace paměti, chyba při otvírání souboru s řídicím programem atd.).

10.2 Paměťový model

Hodnoty během interpretace nejčastěji ukládáme do pojmenovaných proměnných, které jsou sdružovány do tzv. rámců, což jsou v podstatě slovníky proměnných s jejich hodnotami. IFJcode25 nabízí tři druhy rámců:

- globální, značíme GF (Global Frame), který je na začátku interpretace automaticky inicializován jako prázdný; slouží pro ukládání globálních proměnných;
- lokální, značíme LF (Local Frame), který je na začátku nedefinován a odkazuje na vrcholový/aktuální rámec na zásobníku rámců; slouží pro ukládání lokálních proměnných funkcí (zásobník rámců lze s výhodou využít při zanořeném či rekurzivním volání funkcí);
- dočasný, značíme TF (Temporary Frame), který slouží pro chystání nového nebo úklid starého rámce (např. při volání nebo dokončování funkce), jenž může být přesunut na zásobník rámců a stát se aktuálním lokálním rámcem. Na začátku interpretace je dočasný rámec nedefinovaný.

K překrytým (dříve vloženým) lokálním rámcům v zásobníku rámců nelze přistoupit dříve, než vyjmeme později přidané rámce.

Další možností pro ukládání nepojmenovaných hodnot je datový zásobník využívaný zásobníkovými instrukcemi.

10.3 Datové typy

Interpret IFJcode25 pracuje s typy operandů dynamicky, takže je typ proměnné (resp. paměťového místa) dán obsaženou hodnotou. Není-li řečeno jinak, jsou implicitní konverze zakázány. Interpret podporuje speciální hodnotu/typ nil a čtyři základní datové typy (int, bool, float a string), jejichž rozsahy i přesnosti jsou kompatibilní s jazykem IFJ25.

Zápis každé konstanty v IFJcode25 se skládá ze dvou částí oddělených zavináčem (znak `@`; bez bílých znaků), označení typu konstanty (int, bool, float, string, nil) a samotné konstanty (číslo, literál, nil). Příklady: `float@0x1.2666666666666p+0`, `bool@true`, `nil@nil` nebo `int@-5`.

Typ int reprezentuje 64-bitové celé číslo (rozsah C-long long int). Typ bool reprezentuje pravdivostní hodnotu (true nebo false). Typ float popisuje desetinné číslo (rozsah C-double) a v případě zápisu konstant používejte v jazyce C formátovací řetězec `'%a'` pro funkci **`printf`**. Literál pro

typ string je v případě konstanty zapsán jako sekvence tisknutelných ASCII znaků (vyjma bílých znaků, mřížky # a zpětného lomítka \) a escape sekvencí, takže není ohraničen uvozovkami. Escape sekvence, která je nezbytná pro znaky s ASCII kódem 000–032, 035 a 092, je tvaru *xyz*, kde *xyz* je dekadické číslo v rozmezí 000–255 složené právě ze tří číslic; např. konstanta

```
string@retezec\032s\032lomitkem\032\092\032a\010novym\035radkem
```

reprezentuje řetězec

```
retezec s lomitkem \ a
novym#radkem
```

Pokus o práci s neexistující proměnnou (čtení nebo zápis) vede na chybu 54. Pokus o čtení hodnoty neinicializované proměnné vede na chybu 56. Pokus o interpretaci instrukce s operandy nevhodných typů dle popisu dané instrukce vede na chybu 53.

10.4 Instrukční sada

U popisu instrukcí sázíme operační kód tučně a operandy zapisujeme pomocí neterminálních symbolů (případně číslovaných) v úhlových závorkách. Neterminál ⟨*var*⟩ značí proměnnou, ⟨*symb*⟩ konstantu nebo proměnnou, ⟨*label*⟩ značí návěští. Identifikátor proměnné se skládá ze dvou částí oddělených zavináčem (znak `@`; bez bílých znaků), označení rámce LF, TF nebo GF a samotného jména proměnné (sekvence libovolných alfanumerických a speciálních znaků bez bílých znaků začínající písmenem nebo speciálním znakem, kde speciální znaky jsou: `_`, `-`, `$`, `&`, `%`, `*`, `!`, `?`). Např. `GF@_x` značí proměnnou `_x` uloženou v globálním rámci.

Na zápis návěští se vztahují stejná pravidla jako na jméno proměnné (tj. část identifikátoru za zavináčem).

Instrukční sada nabízí instrukce pro práci s proměnnými v rámcích, různé skoky, operace s datovým zásobníkem, aritmetické, řetězcové, logické a relační operace, dále také konverzní, vstupně/výstupní a ladicí instrukce.

10.4.1 Práce s rámci, volání funkcí

MOVE ⟨*var*⟩ ⟨*symb*⟩ — Přiřazení hodnoty do proměnné

Zkopíruje hodnotu ⟨*symb*⟩ do ⟨*var*⟩. Např. `MOVE LF@par GF@var` provede zkopírování hodnoty proměnné `var` v globálním rámci do proměnné `par` v lokálním rámci.

CREATEFRAME — Vytvoř nový dočasný rámec

Vytvoří nový dočasný rámec a zahodí případný obsah původního dočasného rámce.

PUSHFRAME — Přesun dočasného rámce na zásobník rámců

Přesuň TF na zásobník rámců. Rámec bude k dispozici přes LF a překryje původní rámce na zásobníku rámců. TF bude po provedení instrukce nedefinován a je třeba jej před dalším použitím vytvořit pomocí CREATEFRAME. Pokus o přístup k nedefinovanému rámci vede na chybu 55.

POPFRAME — Přesun aktuálního rámce do dočasného

Přesuň vrcholový rámec LF ze zásobníku rámců do TF. Pokud žádný rámec v LF není k dispozici, dojde k chybě 55.

DEFVAR ⟨*var*⟩ Definuj novou proměnnou v rámci

Definuje proměnnou v určeném rámci dle ⟨*var*⟩. Tato proměnná je zatím neinicializovaná a bez určení typu, který bude určen až přiřazením nějaké hodnoty.

CALL ⟨*label*⟩ Skok na návěští s podporou návratu

Uloží inkrementovanou aktuální pozici z interního čítače instrukcí do zásobníku volání a provede skok na zadané návěští (případnou přípravu rámce musí zajistit jiné instrukce).

RETURN Návrat na pozici uloženou instrukcí CALL

Vyjme pozici ze zásobníku volání a skočí na tuto pozici nastavením interního čítače instrukcí (úklid lokálních rámců musí zajistit jiné instrukce). Provedení instrukce při prázdném zásobníku volání vede na chybu 56.

10.4.2 Práce s datovým zásobníkem

Operační kód zásobníkových instrukcí je zakončen písmenem „S“. Zásobníkové instrukce načítají chybějící operandy z datového zásobníku a výslednou hodnotu operace ukládají zpět na datový zásobník.

PUSHS ⟨*symb*⟩ Vlož hodnotu na vrchol datového zásobníku

Uloží hodnotu ⟨*symb*⟩ na datový zásobník.

POPS ⟨*var*⟩ Vyjmi hodnotu z vrcholu datového zásobníku

Není-li zásobník prázdný, vyjme z něj hodnotu a uloží ji do proměnné ⟨*var*⟩, jinak dojde k chybě 56.

CLEARS Vymazání obsahu celého datového zásobníku

Pomocná instrukce, která smaže celý obsah datového zásobníku, aby neobsahoval zapomenuté hodnoty z předchozích výpočtů.

10.4.3 Aritmetické, relační, booleovské a konverzní instrukce

V této sekci jsou popsány tříadresné i zásobníkové verze instrukcí pro klasické operace pro výpočet výrazu. Zásobníkové verze instrukcí z datového zásobníku vybírají operandy se vstupními hodnotami dle popisu tříadresné instrukce od konce (tj. typicky nejprve ⟨$symb_2$⟩ a poté ⟨$symb_1$⟩).

ADD ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Součet dvou číselných hodnot

Sečte ⟨$symb_1$⟩ a ⟨$symb_2$⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨*var*⟩.

SUB ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Odečítání dvou číselných hodnot

Odečte ⟨$symb_2$⟩ od ⟨$symb_1$⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨*var*⟩.

MUL ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Násobení dvou číselných hodnot

Vynásobí ⟨$symb_1$⟩ a ⟨$symb_2$⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨*var*⟩.

DIV ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Dělení dvou desetinných hodnot

Podělí hodnotu ze ⟨$symb_1$⟩ druhou hodnotou ze ⟨$symb_2$⟩ (oba musí být typu float) a výsledek přiřadí do proměnné ⟨*var*⟩ (též typu float). Dělení nulou způsobí chybu 57.

IDIV ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Dělení dvou celočíselných hodnot

Celočíselně podělí hodnotu ze ⟨$symb_1$⟩ druhou hodnotou ze ⟨$symb_2$⟩ (musí být oba typu int) a výsledek (zaokrouhlený k $-\infty$) přiřadí do proměnné ⟨*var*⟩ typu int. Dělení nulou způsobí chybu 57.

ADDS/SUBS/MULS/DIVS/IDIVS Zásobníkové verze instrukcí ADD, SUB, MUL, DIV a IDIV

LT/GT/EQ ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Relační operátory menší, větší, rovno

Instrukce vyhodnotí relační operátor mezi ⟨$symb_1$⟩ a ⟨$symb_2$⟩ (stejného typu; int, bool, float nebo string) a do booleovské proměnné ⟨*var*⟩ zapíše false při neplatnosti nebo true v případě platnosti odpovídající relace. Řetězce jsou porovnávány lexikograficky a false je menší než true. Pro výpočet neostrých nerovností lze použít AND/OR/NOT. S operandem typu nil (druhý operand je libovolného typu) lze porovnávat pouze instrukcí EQ, jinak chyba 53.

LTS/GTS/EQS Zásobníková verze instrukcí LT/GT/EQ

AND/OR/NOT ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Základní booleovské operátory

Aplikuje konjunkci (logické A)/disjunkci (logické NEBO) na operandy typu bool ⟨$symb_1$⟩ a ⟨$symb_2$⟩ nebo negaci na ⟨$symb_1$⟩ (NOT má pouze 2 operandy) a výsledek typu bool zapíše do ⟨*var*⟩.

ANDS/ORS/NOTS Zásobníková verze instrukcí AND, OR a NOT

INT2FLOAT ⟨*var*⟩ ⟨*symb*⟩ Převod celočíselné hodnoty na desetinnou

Převede celočíselnou hodnotu ⟨*symb*⟩ na desetinné číslo a uloží jej do ⟨*var*⟩.

FLOAT2INT ⟨*var*⟩ ⟨*symb*⟩ Převod desetinné hodnoty na celočíselnou (oseknutí)

Převede desetinnou hodnotu ⟨*symb*⟩ na celočíselnou oseknutím desetinné části a uloží ji do ⟨*var*⟩.

INT2CHAR ⟨*var*⟩ ⟨*symb*⟩ Převod celého čísla na znak

Číselná hodnota ⟨*symb*⟩ je dle ASCII převedena na znak, který tvoří jednoznakový řetězec přiřazený do ⟨*var*⟩. Je-li ⟨*symb*⟩ mimo interval $\langle 0; 255\rangle$, dojde k chybě 58.

STRI2INT ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Ordinální hodnota znaku

Do ⟨*var*⟩ uloží ordinální hodnotu znaku (dle ASCII) v řetězci ⟨$symb_1$⟩ na pozici ⟨$symb_2$⟩ (indexováno od nuly). Indexace mimo daný řetězec vede na chybu 58.

FLOAT2STR ⟨*var*⟩ ⟨*symb*⟩ Převod desetinné hodnoty na řetězec

Převede desetinnou hodnotu ⟨*symb*⟩ na řetězec a uloží jej do ⟨*var*⟩. Hodnota je převedena v souladu s chováním vestavěné funkce `Ifj.str` jazyka IFJ25.

INT2STR ⟨*var*⟩ ⟨*symb*⟩ Převod celočíselné hodnoty na řetězec

Převede celočíselnou hodnotu ⟨*symb*⟩ na řetězec a uloží jej do ⟨*var*⟩. Hodnota je převedena v souladu s chováním vestavěné funkce `Ifj.str` jazyka IFJ25.

INT2FLOATS/FLOAT2INTS/ INT2CHARS/STRI2INTS/ FLOAT2STRS/INT2STRS — Zásobníkové verze konverzních instrukcí

10.4.4 Vstupně-výstupní instrukce

READ ⟨*var*⟩ ⟨*type*⟩ — Načtení hodnoty ze standardního vstupu

Načte jednu hodnotu dle zadaného typu ⟨*type*⟩ ∈ {int, float, string, bool} a uloží tuto hodnotu do proměnné ⟨*var*⟩. Podporované hodnoty typu float odpovídají funkci `strtod` jazyka C (kromě nan, inf). Formát hodnot je kompatibilní s chováním vestavěných funkcí **`Ifj.read_str`**, **`Ifj.read_num`** (a příp. **`Ifj.read_bool`**) jazyka IFJ25.

WRITE ⟨*symb*⟩ — Výpis hodnoty na standardní výstup

Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je v souladu s vestavěným příkazem **`Ifj.write`** jazyka IFJ25 (pozor, hodnota typu float je vždy vypsána pomocí formátovacího řetězce `%a`, hodnota typu int je vypsána pomocí řetězce `%d`).

10.4.5 Práce s řetězci

CONCAT ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ — Konkatenace dvou řetězců

Do proměnné ⟨*var*⟩ uloží řetězec vzniklý konkatenací dvou řetězcových operandů ⟨$symb_1$⟩ a ⟨$symb_2$⟩ (jiné typy nejsou povoleny).

STRLEN ⟨*var*⟩ ⟨*symb*⟩ — Zjisti délku řetězce

Zjistí délku řetězce v ⟨*symb*⟩ a délka je uložena jako celé číslo do ⟨*var*⟩.

GETCHAR ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ — Vrať znak řetězce

Do ⟨*var*⟩ uloží řetězec z jednoho znaku v řetězci ⟨$symb_1$⟩ na pozici ⟨$symb_2$⟩ (indexováno celým číslem od nuly). Indexace mimo daný řetězec vede na chybu 58.

SETCHAR ⟨*var*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ — Změň znak řetězce

Zmodifikuje znak řetězce uloženého v proměnné ⟨*var*⟩ na pozici ⟨$symb_1$⟩ (indexováno celočíselně od nuly) na znak v řetězci ⟨$symb_2$⟩ (první znak, pokud obsahuje ⟨$symb_2$⟩ více znaků). Výsledný řetězec je opět uložen do ⟨*var*⟩. Při indexaci mimo řetězec ⟨*var*⟩ nebo v případě prázdného řetězce v ⟨$symb_2$⟩ dojde k chybě 58.

10.4.6 Práce s typy

TYPE ⟨*var*⟩ ⟨*symb*⟩ — Zjisti typ daného symbolu

Dynamicky zjistí typ symbolu ⟨*symb*⟩ a do ⟨*var*⟩ zapíše řetězec značící tento typ (int, bool, float, string nebo nil). Je-li ⟨*symb*⟩ neinicializovaná proměnná, označí její typ prázdným řetězcem.

ISINT ⟨*var*⟩ ⟨*symb*⟩ — Urči celočíselnost symbolu

Určí, jestli je číslo v symbolu ⟨*symb*⟩ celé (tj. typu int nebo typu float s nulovou desetinnou částí), a výsledek typu bool zapíše do ⟨*var*⟩. Operand ⟨*symb*⟩ musí být typu int nebo float.

TYPES/ISINTS — Zásobníková verze TYPE, ISINT

10.4.7 Instrukce pro řízení toku programu

Neterminál ⟨*label*⟩ označuje návěští, které slouží pro označení pozice v kódu IFJcode25. V případě skoku na neexistující návěští dojde k chybě 52.

LABEL ⟨*label*⟩ Definice návěští
Speciální instrukce označující pomocí návěští ⟨*label*⟩ důležitou pozici v kódu jako potenciální cíl libovolné skokové instrukce. Pokus o redefinici existujícího návěští je chybou 52.

JUMP ⟨*label*⟩ Nepodmíněný skok na návěští
Provede nepodmíněný skok na zadané návěští ⟨*label*⟩.

JUMPIFEQ ⟨*label*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Podmíněný skok na návěští při rovnosti
Pokud jsou ⟨$symb_1$⟩ a ⟨$symb_2$⟩ stejného typu nebo je některý operand nil (jinak chyba 53) a zároveň se jejich hodnoty rovnají, tak provede skok na návěští ⟨*label*⟩.

JUMPIFNEQ ⟨*label*⟩ ⟨$symb_1$⟩ ⟨$symb_2$⟩ Podmíněný skok na návěští při nerovnosti
Jsou-li ⟨$symb_1$⟩ a ⟨$symb_2$⟩ stejného typu nebo je některý operand nil (jinak chyba 53), ale různé hodnoty, tak provede skok na návěští ⟨*label*⟩.

JUMPIFEQS/JUMPIFNEQS ⟨*label*⟩ Zásobníková verze JUMPIFEQ, JUMPIFNEQ
Zásobníkové skokové instrukce mají i jeden operand mimo datový zásobník, a to návěští ⟨*label*⟩, na které se případně provede skok.

EXIT ⟨*symb*⟩ Ukončení interpretace s návratovým kódem
Ukončí vykonávání programu a ukončí interpret s návratovým kódem ⟨*symb*⟩, kde ⟨*symb*⟩ je celé číslo v intervalu 0 až 49 (včetně). Nevalidní celočíselná hodnota ⟨*symb*⟩ vede na chybu 57.

10.4.8 Ladicí instrukce

BREAK Výpis stavu interpretu na `stderr`
Na standardní chybový výstup (stderr) vypíše stav interpretu v danou chvíli (tj. během vykonávání této instrukce). Stav se mimo jiné skládá z pozice v kódu, výpisu globálního, aktuálního lokálního a dočasného rámce a počtu již vykonaných instrukcí.

DPRINT ⟨*symb*⟩ Výpis hodnoty na `stderr`
Vypíše zadanou hodnotu ⟨*symb*⟩ na standardní chybový výstup (stderr). Výpisy touto instrukcí bude možné vypnout pomocí volby interpretu (viz nápověda interpretu).

---

11 Pokyny ke způsobu vypracování a odevzdání

Tyto důležité informace nepodceňujte, neboť projekty bude částečně opravovat automat a nedodržení těchto pokynů povede k tomu, že automat daný projekt nebude schopen přeložit, zpracovat a ohodnotit, což může vést až ke ztrátě všech bodů z projektu!

11.1 Obecné informace

Za celý tým odevzdá projekt vedoucí. Všechny odevzdané soubory budou zkomprimovány programem ZIP, TAR+GZIP nebo TAR+BZIP do jediného archivu, který se bude jmenovat `xlogin99.zip`, `xlogin99.tgz`, nebo `xlogin99.tbz`, kde místo zástupného řetězce `xlogin99` použijte školní přihlašovací jméno (VUT login) **vedoucího** týmu. Archiv nesmí obsa-

hovat adresářovou strukturu ani speciální či spustitelné soubory. Názvy všech souborů budou obsahovat pouze písmena **`a-z`**[11], číslice, tečku a podtržítko (ne mezery!).

Celý projekt je třeba odevzdat v daném termínu (viz výše). Pokud tomu tak nebude, je projekt považován za neodevzdaný. Stejně tak, pokud se bude jednat o plagiátorství jakéhokoliv druhu, je projekt hodnocený nula body, navíc v IFJ ani v IAL nebude udělen zápočet a bude zváženo zahájení disciplinárního řízení.

Vždy platí, že je třeba při řešení problémů aktivně a konstruktivně komunikovat nejen uvnitř týmu, ale občas i se cvičícím. Při komunikaci uvádějte login vedoucího a případně jméno týmu.

11.2 Dělení bodů

Odevzdaný archiv bude povinně obsahovat soubor **`rozdeleni`**, ve kterém zohledníte dělení bodů mezi jednotlivé členy týmu (i při požadavku na rovnoměrné dělení). Na každém řádku je pro jednoho člena týmu uvedeno:

VUT ID (osobní číslo), **dvojtečka**, odpovídající FIT login (xlogin), **dvojtečka**, požadovaný celočíselný počet procent bodů bez uvedení znaku %

Každý řádek (i poslední) je poté ihned ukončen jedním **unixovým znakem nového řádku** (známý také jako „line feed“, ⟨LF⟩, ASCII hodnota 10 nebo **`\n`**, tj. unixové ukončení řádku LF, **ne** windowsovské CRLF!).

Obsah souboru bude tedy vypadat například takto (LF zastupuje unixové odřádkování):

```
123456:xnovaka01:30LF
117890:xnovakh02:40LF
124567:xnovako03:30LF
115678:xnovakj04:00LF
```

Součet všech procent musí být roven 100. V případě chybného celkového součtu všech procent bude použito rovnoměrné rozdělení. Formát odevzdaného souboru musí být správný a obsahovat všechny registrované členy týmu (i ty hodnocené 0 %).

Vedoucí týmu je před odevzdáním projektu povinen celý tým informovat o rozdělení bodů. Každý člen týmu je navíc povinen rozdělení bodů zkontrolovat po odevzdání do StudIS a případně rozdělení bodů reklamovat u cvičícího ještě před obhajobou projektu.

12 Požadavky na řešení

Kromě požadavků na implementaci a dokumentaci obsahuje tato sekce i několik rad pro zdárné řešení tohoto projektu. Upozorňujeme, že **projekt bude hodnocen pouze jako funkční celek, a nikoli jako soubor separátních, společně nekooperujících modulů.**

Podrobnější informace o způsobu automatického i manuálního hodnocení, jakož i různé tipy pro práci s jazykem C, najdete v Moodle IFJ na stránce **„Jak se projekty hodnotí, na co si dát pozor“**. Při týmové práci dbejte na dobrou čitelnost kódu a **konzistenci** (způsob pojmenovávání identifikátorů, způsob formátování, …) napříč všemi členy týmu! Používejte standardní techniky zajištění kvality kódu (verzování, code review, …).

[11] Po přejmenování změnou velkých písmen na malá musí být všechny názvy souborů stále unikátní.

12.1 Závazné metody pro implementaci překladače

Pro implementaci jednotlivých částí překladače **povinně** využijte následující metody. Všechny budou probírány na přednáškách předmětu IFJ. Nedodržení těchto metod bude penalizováno značnou ztrátou bodů!

1. Lexikální analýzu implementujte (v principu) jako **konečný automat**.
2. Při konstrukci syntaktické analýzy založené na LL gramatice (vše kromě výrazů) využijte buď **metodu rekurzivního sestupu** (doporučeno), nebo prediktivní analýzu řízenou LL tabulkou.
3. Výrazy (vč. pravdivostních) zpracujte pouze pomocí **precedenční syntaktické analýzy**, která odpovídá výkladu v přednáškách.

Implementaci proveďte **v jazyce C** (čímž úmyslně omezujeme možnosti použití objektově orientované implementace). Návrh implementace překladače je zcela v režii řešitelských týmů. Není dovoleno otevírat síťová spojení nebo jinak komunikovat po síti, spouštět další procesy a vytvářet nové či modifikovat existující soubory (ani v adresáři `/tmp`).

12.2 Implementace tabulky symbolů v souboru `symtable.c`

Implementaci tabulky symbolů (dle varianty zadání) proveďte dle přístupů probíraných v předmětu IAL a umístěte ji do souboru `symtable.c`. Pokud se rozhodnete o odlišný způsob implementace, vysvětlete v dokumentaci důvody, které vás k tomu vedly, a uveďte zdroje, ze kterých jste čerpali.

12.3 Textová část řešení

Součástí řešení bude dokumentace vypracovaná ve formátu PDF a uložená v jediném souboru **`dokumentace.pdf`**. Jakýkoliv jiný než předepsaný formát dokumentace bude ignorován, což povede ke ztrátě bodů za dokumentaci. Dokumentace bude vypracována v českém, slovenském nebo anglickém jazyce v rozsahu cca 3–5 stran A4.

V dokumentaci popisujte návrh (části překladače a předávání informací mezi nimi), implementaci (použité datové struktury, tabulku symbolů, generování kódu), vývojový cyklus, způsob práce v týmu, speciální použité techniky a algoritmy a různé odchylky od přednášené látky či tradičních přístupů. Nezapomínejte také citovat literaturu a uvádět reference na čerpané zdroje včetně správné citace převzatých částí (obrázky, magické konstanty, vzorce). Nepopisujte záležitosti obecně známé či přednášené na naší fakultě.

Dokumentace musí povinně obsahovat (povinné tabulky a diagramy se nezapočítávají do doporučeného rozsahu):

- 1. strana: jména, příjmení a přihlašovací jména řešitelů (označení vedoucího) + údaje o rozdělení bodů, identifikaci vaší varianty zadání ve tvaru „Tým *login_vedoucího*, varianta *X*“ a výčet identifikátorů implementovaných rozšíření.
- Rozdělení práce mezi členy týmu (uveďte kdo a jak se podílel na jednotlivých částech projektu; povinně zdůvodněte odchylky od rovnoměrného rozdělení bodů).
- Diagram konečného automatu, který specifikuje lexikální analyzátor.
- LL gramatiku, LL tabulku a precedenční tabulku, podle kterých jste implementovali váš syntaktický analyzátor.

- Stručný popis členění implementačního řešení včetně názvů souborů, kde jsou jednotlivé části včetně povinných implementovaných metod překladače k nalezení.

Dokumentace nesmí:

- obsahovat kopii zadání či text, obrázky[12] nebo diagramy, které nejsou vaše původní (kopie z přednášek, sítě, WWW, …),
- být založena pouze na výčtu a obecném popisu jednotlivých použitých metod (jde o váš vlastní přístup k řešení; a proto dokumentujte postup, kterým jste se při řešení ubírali; překážkách, se kterými jste se při řešení setkali; problémech, které jste řešili a jak jste je řešili; atd.).

V rámci dokumentace bude rovněž vzat v úvahu stav kódu jako jeho čitelnost, srozumitelnost a dostatečné, ale nikoli přehnané komentáře.

12.4 Programová část řešení

Programová část řešení bude vypracována v jazyce C bez použití generátorů (např. lex, flex, yacc, Bison či jiných podobného typu) a musí být přeložitelná překladačem `gcc`. Při hodnocení budou projekty překládány na školním serveru `merlin`. Počítejte tedy s touto skutečností (především pokud budete projekt psát pod jiným OS). Pokud projekt nepůjde přeložit či nebude správně pracovat kvůli použití funkce nebo nějaké nestandardní implementační techniky závislé na OS, nebude projekt hodnocený. Ve sporných případech bude vždy za platný považován výsledek překladu a testování na serveru `merlin` bez použití jakýchkoliv dodatečných nastavení (proměnné prostředí, …).

Součástí řešení bude soubor `Makefile` sloužící pro překlad projektu pomocí příkazu `make`. Pokud soubor pro sestavení cílového programu nebude obsažen nebo se na jeho základě nepodaří sestavit cílový program, nebude projekt hodnocený! Jméno cílového programu není rozhodující, bude přejmenován automaticky. Binární soubor (přeložený překladač) v žádném případě do archivu nepřikládejte!

Úvod **všech** zdrojových textů musí obsahovat zakomentovaný název projektu, přihlašovací jména a jména studujících, kteří/které se na daném souboru skutečně autorsky podíleli/y.

Veškerá chybová hlášení vzniklá v průběhu činnosti překladače budou vždy vypisována na standardní chybový výstup. Veškeré texty tištěné řídicím programem budou vypisovány na standardní výstup, pokud není explicitně řečeno jinak. Kromě chybových/ladicích hlášení vypisovaných na standardní chybový výstup nebude generovaný mezikód přikazovat výpis žádných znaků či dokonce celých textů, které nejsou přímo předepsány řídicím programem.

Základní testování bude probíhat pomocí automatu, který bude postupně vaším překladačem kompilovat sadu testovacích příkladů, výsledky interpretovat naším interpretem jazyka IFJcode25 a porovnávat produkované výstupy na standardní výstup s výstupy očekávanými. Pro porovnání výstupů bude použit program `diff` (viz `info diff`). Proto jediný neočekávaný znak, který bude při hodnotící interpretaci vámi vygenerovaného kódu svévolně vytisknut, povede k nevyhovujícímu hodnocení aktuálního výstupu, a tím snížení bodového hodnocení celého projektu.

12.5 Jak postupovat při řešení projektu

Instalace překladače `gcc` není třeba, pokud máte již instalovaný jiný překladač jazyka C, avšak nesmíte v tomto překladači využívat vlastnosti, které `gcc` nepodporuje. Před použitím nějaké

[12] Vyjma obyčejného loga fakulty na úvodní straně.

vyspělé konstrukce je dobré si ověřit, že jí disponuje i překladač `gcc` na serveru Merlin. Po vypracování je též vhodné vše ověřit na serveru Merlin, aby při překladu a hodnocení projektu vše proběhlo bez problémů. V *Moodle* IFJ bude odkazován skript `is_it_ok.sh` na kontrolu většiny formálních požadavků odevzdávaného archivu, který doporučujeme využít.

Teoretické znalosti potřebné pro vytvoření projektu získáte během semestru na přednáškách, Moodle a diskuzním fóru IFJ. Postupuje-li vaše realizace projektu rychleji než probírání témat na přednášce, doporučujeme využít samostudium (viz zveřejněné záznamy z minulých let a detailnější pokyny na Moodle IFJ). Je nezbytné, aby na řešení projektu spolupracoval celý tým. Návrh překladače, základních rozhraní a rozdělení práce lze vytvořit již v první čtvrtině semestru. Je dobré, když se celý tým domluví na pravidelných schůzkách a komunikačních kanálech, které bude během řešení projektu využívat (instant messaging, video konference, verzovací systém, štábní kulturu atd.).

Situaci, kdy je projekt ignorován částí týmu, lze řešit prostřednictvím souboru `rozdeleni` a extrémní případy řešte přímo se cvičícími co nejdříve. Je ale nutné, abyste si vzájemně (nespoléhejte pouze na vedoucího), nejlépe na pravidelných schůzkách týmu, **ověřovali skutečný pokrok** v práci na projektu a případně včas přerozdělili práci.

Pokud využíváte editor Visual Studio Code (nebo nějaký jeho derivát), doporučujeme využít rozšíření pro zvýrazňování syntaxe v jazyce Wren a rozšíření pro podporu cílového jazyka IFJcode25.

Maximální počet bodů získatelný na jednu osobu za programovou implementaci je **23** včetně bonusových bodů za rozšíření projektu.

Nenechávejte řešení projektu až na poslední týden. Projekt je tvořen z několika částí (např. lexikální analýza, syntaktická analýza, sémantická analýza, tabulka symbolů, generování mezikódu, dokumentace, testování!) a dimenzován tak, aby jednotlivé části bylo možno navrhnout a implementovat již v průběhu semestru na základě znalostí získaných na přednáškách předmětů IFJ a IAL a samostudiem na Moodle a diskuzním fóru předmětu IFJ.

12.6 Pokusné odevzdání

Pro zvýšení motivace pro včasné vypracování projektu nabízíme koncept nepovinného pokusného odevzdání. Výměnou za pokusné odevzdání do uvedeného termínu (několik týdnů před finálním termínem) dostanete zpětnou vazbu v podobě procentuálního hodnocení aktuální kvality vašeho projektu.

Pokusné odevzdání bude relativně rychle vyhodnoceno automatickými testy. Zašleme vám informace o procentuální správnosti stěžejních částí pokusně odevzdaného projektu z hlediska části automatických testů (tj. nebude se jednat o finální hodnocení, proto nebudou sdělovány ani body). Výsledky nejsou nijak bodovány, a proto nebudou individuálně sdělovány žádné detaily k chybám v zaslaných projektech, jako je tomu u finálního termínu. Využití pokusného termínu není povinné, ale jeho nevyužití může být vzato v úvahu jako přitěžující okolnost v případě různých reklamací.

Formální požadavky na pokusné odevzdání jsou totožné s požadavky na finální termín a odevzdání se bude provádět do speciální aktivity „Projekt – Pokusné odevzdání“ předmětu IFJ. Není nutné zahrnout dokumentaci, která spolu s rozšířeními pokusně vyhodnocena nebude. Pokusně odevzdává nejvýše jeden člen týmu (nejlépe vedoucí), který se na zadání v rámci pokusného odevzdání registruje ve StudIS, odevzdává a následně obdrží jeho vyhodnocení, o kterém informuje zbytek týmu.

13 Registrovaná rozšíření

V případě implementace některých registrovaných rozšíření bude odevzdaný archiv obsahovat soubor **`rozsireni`**, ve kterém uvedete na každém řádku identifikátor jednoho implementovaného rozšíření (řádky jsou opět ukončeny znakem ⟨LF⟩).

V průběhu řešení (do stanoveného termínu) bude postupně (případně i na váš popud) aktualizován ceník rozšíření a identifikátory rozšíření projektu (viz Moodle a diskuzní fórum k předmětu IFJ). V něm budou uvedena hodnocená rozšíření projektu, za která lze získat prémiové body. Cvičícím můžete během semestru zasílat návrhy na dosud neuvedená rozšíření, která byste chtěli navíc implementovat. Cvičící rozhodnou o přijetí/nepřijetí rozšíření a hodnocení rozšíření dle jeho náročnosti včetně přiřazení unikátního identifikátoru. Body za implementovaná rozšíření se počítají do bodů za programovou implementaci, takže stále platí získatelné maximum 23 bodů.

Popis rozšíření vždy začíná jeho identifikátorem a maximálním počtem možných získaných bodů. Většina těchto rozšíření je založena na dalších vlastnostech jazyka Wren, nemusí však jazyku Wren odpovídat v celém jeho rozsahu. Podrobnější informace lze získat ze specifikace jazyka Wren. Pokud váháte např. s určením vhodného chybového kódu, využijte diskuzní fórum (obvykle budeme akceptovat všechny možnosti, které „dávají smysl“). Do dokumentace je potřeba (kromě zkratky na úvodní stranu) také popsat způsob, jakým jsou implementovaná rozšíření řešena.

Tip: Pokud s podporou rozšíření budete počítat hned od začátku, jejich implementace obvykle nezabere příliš mnoho času ani úsilí navíc, což za bonusové body stojí. Většina rozšíření vyžaduje úpravu gramatiky, tudíž je jejich doplňování do už hotového překladače spíše nepraktické.

Poznámka: V rozšířeních testujeme hlavně jádro funkcionality, nezaměřujeme se na krajní případy. Alespoň část bodů lze získat i za nekompletní implementaci rozšíření.

FUNEXP (+1,5 bodu) Volání funkce může být součástí výrazu. Výraz může být parametrem ve volání funkce.

EXTSTAT (+0,25 bodu) Podporujte příkaz definice lokální proměnné rožšířený o možnost hned nastavit hodnotu; příkaz volání funkce s implicitní dekalarcí proměnné; a příkaz návratu z funkce bez uvedení výrazu:

`var` *idl* **`=`** *výraz*
`var` *idl* **`=`** *id_funkce* **`(`***seznam_vstupních_parametrů***`)`**
`var` *idl* **`=`** *id_fce_getter*
`return`

BOOLTHEN (+1,0 bodu) Podpora typu **`Bool`**, booleovských hodnot **`true`** a **`false`**, booleovských výrazů včetně kulatých závorek. Podporujte přiřazování výsledku booleovského výrazu do proměnné. Podporujte negační unární operátor **`!`**. Pravdivostní hodnoty lze porovnávat jen operátory **`==`** a **`!=`**.

Podporujte také booleovské operátory **`&&`** a **`||`**, jejichž priorita, asociativita i sémantika odpovídá jazyku Wren. (Pozorně nastudujte jejich sémantiku – povšimněte si, že nepracují pouze s hodnotami typu **`Bool`**!)

Podporujte také zkrácený podmíněný příkaz (bez části **`else`**) a rozšířený podmíněný příkaz **`if`** – **`else if`** – **`else`**.

Hodnoty typu **`Bool`** podporujte také ve vestavěných funkcích **`Ifj.write`** a **`Ifj.str`**. Podporujte také novou vestavěnou funkci **`Ifj.read_bool`** pro načítání těchto hodnot (v souladu s chováním instrukce READ rozpoznává pouze řetězce **`true`** a **`false`**, jinak vrací **`null`**).

CYCLES (+1,0 bodu) Podporujte cyklus **`for`**:

`for (` *id* **`in`** *výraz_rozsahu* **`)`** *blok*

Výraz_rozsahu je zde výraz, který obsahuje právě jeden výskyt operátoru exkluzivního rozsahu '`...`' nebo operátoru inkluzivního rozsahu '`..`'. Tyto operátory mají prioritu a asociativitu odpovídající jazyku Wren. Na levé i pravé straně výrazu musí být celočíselná hodnota typu **`Num`** (jinak chyba 6 / 26). Není nutné, aby bylo možné výsledek operátoru rozsahu uložit do proměnné (v jazyce Wren použitím operátoru vzniká hodnota typu **`Range`** – zde to můžete implementovat obdobně nebo úplně jinak).

Dále v cyklu typu **`for`** i **`while`** podporujte klíčová slova **`break`** a **`continue`**.

OPERATORS (+0,25 bodu) Podporujte unární operátor **`-`** (unární mínus), vč. možnosti používat záporné číselné literály. Podporujte ternární operátor **`?:`** s běžnou sémantikou (odpovídající jazyku Wren).

ONELINEBLOCK (+0,25 bodu) Podporujte *jednořádkové bloky*, které **ne**mají za otevírací složenou závorkou znak nového řádku. Je možné použít je všude, kde se očekává blok, mají však speciální význam, pokud tvoří tělo funkce. Mohou obsahovat nejvýše jeden výraz (pozor, ne příkaz), který pak automaticky určí návratovou hodnotu bloku. Za výrazem nenásleduje znak nového řádku:

`{` *výraz* **`}`**

Následující dvě funkce tedy budou mít stejnou návratovou hodnotu:

```
static func1(x) {
    return x + 1
}
static func2(x) { x + 1 }
```

Podporujte i prázdné jednořádkové bloky (tedy pouze pár složených závorek na jednom řádku). Návratovou hodnotou takového bloku je hodnota **`null`**.

STATICAN (+1,75 bodu) Provádějte statickou analýzu typů ve výrazech s proměnnými nebo gettery a při předávání proměnných nebo getterů jako parametrů funkcí. Pokud implementujete i FUNEXP, analýza bude zahrnovat i volání funkcí uvnitř výrazů. Překlad bude končit s typovou chybou 6 vždy, pokud je typová kolize ve všech případech nevyhnutelná.

Kde to je možné, staticky analyzujte také celočíselnost hodnot typu **`Num`**. Pokud implementujete i rozšíření FUNEXP nebo BOOLTHEN, můžete volitelně přidat vestavěnou funkci **`Ifj.is_int`** a informaci o celočíselnosti propagovat do větvení (viz příklad níže).

V případě funkcí s větvením stačí uvažovat, že funkce může vždy vrátit typ ze všech větví (i pokud jsou některé nedosažitelné). Příklad níže pro jednoduchost uvažuje i rozšíření EXTSTAT.

```
static poly_fun() {  // vraci String nebo Num
    var a = Ifj.read_num()
    if (a > 10) {
        return "retezec"
    } else {
        return a
    }
}

static non_poly_fun() {  // vzdy vraci Num (ale ne nutne celociselny)
    var b = Ifj.read_num()
```

```
    if (b < 20) {
        return b * 2
    } else {
        return b
    }
}

var poly_var = poly_fun() // muze byt String nebo Num
var non_poly_var = non_poly_fun() // jednoznacne Num
var x = 10 + poly_var // neni chyba prekladu (muze byt za behu)
var y = "ahoj" + poly_var // neni chyba prekladu (muze byt za behu)
var z = "nelze" + non_poly_var // je chyba prekladu!

var u = 15 * 10.8 + 49
var v = "ahoj" + " svete"
var w = u + v // je chyba prekladu!

Ifj.ord("ahoj", u) // chyba prekladu - v "u" urcite neni cele cislo
Ifj.ord("ahoj", non_poly_var) // neni chyba prekladu

// Pro odvazne (takto vyzaduje FUNEXP):
static definitely_not_int() {
    var c = Ifj.read_num()
    if (Ifj.is_int(c)) {
        return 0.123
    } else {
        return c
    }
}

Ifj.ord("ahoj", definitely_not_int()) // chyba prekladu
```