ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL

Zbyněk Křivka, Tomáš Kožár, Radim Kocman
email: {krivka, ikozar, kocman}@fit.vut.cz
22. září 2023

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ23.
Informace: diskuzní fórum a Moodle předmětu IFJ.
Pokusné odevzdání: čtvrtek 23. listopadu 2023, 23:59 (nepovinné).
Datum odevzdání: středa 6. prosince 2023, 23:59.
Způsob odevzdání: prostřednictvím StudIS, aktivita „Projekt - Registrace a Odevzdání“.

Hodnocení:

- Do předmětu IFJ získá každý maximálně 25 bodů (15 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).
- Max. 35 % bodů Vašeho individuálního hodnocení základní funkčnosti do předmětu IFJ navíc za tvůrčí přístup (různá rozšíření apod.).
- **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 15 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).

- Pokud bude mít tým dostatek členů, bude mu umožněno si zaregistrovat jednu ze dvou variant zadání (viz sekce 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 IFJ23 a přeloží jej do cílového jazyka IFJcode23 (mezikód). Jestliže proběhne překlad bez chyb, vrací se návratová hodnota `0` (nula). Jestliže došlo k nějaké chybě, 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 hlavička, atp.).
- 3 - sémantická chyba v programu – nedefinovaná funkce.
- 4 - sémantická chyba v programu – špatný počet/typ parametrů u volání funkce či špatný typ návratové hodnoty z funkce.
- 5 - sémantická chyba v programu – použití nedefinované proměnné.
- 6 - sémantická chyba v programu – chybějící/přebývající výraz v příkazu návratu z funkce.
- 7 - sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.
- 8 - sémantická chyba odvození typu – typ proměnné nebo parametru není uveden a nelze odvodit od použitého výrazu.
- 9 - ostatní sémantické chyby.
- 99 - interní chyba překladače tj. neovlivněná vstupním programem (např. chyba alokace paměti atd.).

Překladač bude načítat řídicí program v jazyce IFJ23 ze standardního vstupu a generovat výsledný mezikód v jazyce IFJcode23 (viz kapitola 10) na standardní výstup. Všechna chybová hlášení, varování a ladicí výpisy provádějte na standardní chybový výstup; tj. bude se jednat o konzolovou aplikaci (tzv. filtr) bez grafického uživatelského rozhraní. Pro interpretaci výsledného programu v cílovém jazyce IFJcode23 bude na stránkách 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!

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

3 Popis programovacího jazyka

Jazyk IFJ23 je zjednodušenou podmnožinou jazyka Swift[2], což je moderní a stále populárnější jazyk používaný především při vývoji pro zařízení Apple.

3.1 Obecné vlastnosti a datové typy

V programovacím jazyce IFJ23 **záleží** na velikosti písmen u identifikátorů i klíčových slov (tzv. *case-sensitive*). Jazyk IFJ23 je silně typovaný s podporou jednoduchého odvozování typů.

- *Identifikátor* je definován jako neprázdná posloupnost číslic, písmen (malých i velkých) a znaku podtržítka ('_') začínající písmenem nebo podtržítkem (kromě samotného podtržítka). Jazyk IFJ23 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam[3], a proto se nesmějí vyskytovat jako identifikátory:

  Klíčová slova: **`Double`**, **`else`**, **`func`**, **`if`**, **`Int`**, **`let`**, **`nil`**, **`return`**, **`String`**, **`var`**, **`while`**

- *Identifikátor typu* se skládá z volitelné přípony, což je znak ? (otazník), a klíčového slova pro typ (**`Double`**, **`Int`**, **`String`**). Např. **`Int`** nebo **`String?`**.
- *Celočíselný literál* (rozsah C-int) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě.
- *Desetinný literál* (rozsah C-double) 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 **`"`**) lze zapisovat přímo. Některé další znaky lze zapisovat pomocí escape sekvence: '**`\"`**', '**`\n`**', '**`\r`**', '**`\t`**', '**`\\`**'. Jejich význam se shoduje s odpovídajícími znakovými konstantami jazyka Swift[5]. Na rozdíl od jazyka C nelze escape sekvencí vytvořit chybu – pakliže znaky za zpětným lomítkem neodpovídají žádnému z uvedených vzorů, jsou (včetně lomítka) bez jakýchkoli náhrad součástí řetězce. Expanzi (interpolaci) proměnných či výrazů v řetězcích neuvažujte. Znak v řetězci může být

[2] Online dokumentace: `https://docs.swift.org/swift-book/documentation/the-swift-programming-language/`; na serveru Merlin pod příkazem `swift` je pro studenty k dispozici verze 5.8.1 z 18. května 2023.

[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` v celočíselné části či exponentu jsou ignorovány.

[5] `https://docs.swift.org/swift-book/documentation/the-swift-programming-language/lexicalstructure`

zadán také pomocí escape sekvence '**\u{***dd***}**', kde *dd* je hexadecimální číslo složeno z 1 až 8 hexadecimálních číslic (písmena **A**-**F** mohou být malá i velká), ale testovat budeme hodnoty v rozsahu 0 až 255 .
Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

```
"Ahoj\n\"Sve'te \\\u{22}"
```

reprezentuje řetězec
Ahoj
"Sve'te \". Neuvažujte řetězce, které obsahují vícebajtové znaky kódování Unicode (např. UTF-8).
Podporujte také víceřádkové řetězce, které jsou ohraničeny trojící uvozovek na samostatných řádcích. Odřádkování za počáteční **"""** a před koncovou **"""** není součástí řetězce. Ve víceřádkovém řetězci se mohou vyskytovat znaky, které musely být escapované v klasickém řetězcovém literálu, ale nesmí se tam vyskytovat trojice za sebou jdoucích uvozovek. Odsazení a zahrnování bílých znaků řešte dle dokumentace jazyka Swift[5].

- *Datové typy* pro jednotlivé uvedené literály jsou označeny **Int**, **Double** a **String**. Speciálním případem je hodnota **nil**. Typy se deklarují u proměnných a také v signaturách funkcí. Chybějící deklarace typu proměnné bude překladačem odvozena z přiřazovaného výrazu. Hodnotu **nil** mohou nabývat všechny proměnné, parametry a návratové hodnoty typované s příponou ?, tzv. *typ zahrnující* ***nil***. Z hodnoty **nil** nelze odvodit konkrétní typ.
- *Term* je libovolný literál (celočíselný, desetinný, řetězcový či **nil**) nebo identifikátor.
- Jazyk IFJ23 podporuje *řádkové* a *blokové komentáře* stejně jako v jazyce Swift. Řádkový komentář začíná dvojicí lomítek ('**//**', ASCII hodnota 47) a za komentář je považováno vše, co následuje až do konce řádku.
- Blokový komentář začíná posloupností symbolů '**/***' a je ukončen dvojicí symbolů '***/**'. Podporujte i vnořené blokové komentáře.

4 Struktura jazyka

IFJ23 je strukturovaný programovací jazyk podporující definice modifikovatelných a nemodifikovatelných proměnných a uživatelských funkcí včetně jejich rekurzivního volání. Vstupním bodem prováděného programu je neoznačená nesouvislá sekvence příkazů mezi definicemi uživatelských funkcí, tzv. *hlavní tělo* programu.

4.1 Základní struktura jazyka

Program je tvořen definicemi funkcí prolínajících se s hlavním tělem programu. V hlavním těle lze potom definovat proměnné, používat příkazy přiřazení, větvení, iterace a volání funkcí. V těle uživatelské funkce lze navíc vracet její výsledek.

Před, za i mezi jednotlivými tokeny se může vyskytovat libovolný počet bílých znaků (mezera, tabulátor, komentář a odřádkování), takže jednotlivé konstrukce jazyka IFJ23 lze

zapisovat na jednom či více řádcích. Bílý znak je někdy nutný pro rozlišení unárních a binárních operátorů. Více příkazů na jednom řádku lze zapsat pouze, pokud jsou odděleny nějakým dalším tokenem (ne jen bílým znakem; např. otevírací složená závorka).

Struktura definice uživatelských funkcí a také popis jednotlivých příkazů je v následujících sekcích.

4.1.1 Proměnné

Proměnné jazyka IFJ23 jsou globální a lokální. Globální proměnné jsou definovány pouze v hlavním těle (nikoli ve složeném příkazu v hlavním těle) a mají rozsah v celém programu. Lokální proměnné jsou definované v zanořených blocích hlavního těla a v tělech uživatelských funkcí. Lokální proměnné mají rozsah platnosti od počátku bloku jejich definice až po konec bloku, ve které byly definovány. *Blokem* je libovolná sekvence příkazů zapsána v těle funkce, v rámci větve podmíněného příkazu nebo příkazu cyklu. Blok a jeho podbloky tvoří tzv. *rozsah platnosti*, kde je proměnná dostupná, pokud není v podbloku překryta lokální definicí stejně pojmenované proměnné (překrývat lze i globální proměnné).

Definice nemodifikovatelné proměnné se provádí pomocí příkazu **`let`** a povinném inicializačním výrazu. Definice modifikovatelné (běžné) proměnné provedeme příkazem **`var`**, kde je inicializační výraz nepovinný. Detailní syntaxe bude popsána v sekci 4.4. Každá proměnná musí být definována před jejím použitím, což musí být staticky analyzovatelné, jinak se jedná o chybu 5.

4.2 Deklarace a definice uživatelských funkcí

Definice funkce se skládá z hlavičky a těla funkce. Definice funkce je zároveň její deklarací. Každá funkce musí být definovaná[6], jinak končí analýza chybou 3.

Definice funkce nemusí lexikálně předcházet kódu pro použití této funkce, tzv. *volání funkce*. Uvažujte například vzájemné rekurzivní volání funkcí (tj. funkce `foo` volá funkci `bar`, která opět může volat funkci `foo`). Funkce může být definovaná lexikálně až za jejím použitím, tzv. *voláním funkce* (příkaz volání je definován níže).

Příklad (vzájemná rekurze bez ukončující podmínky):

```
func bar(with param : String) -> String {
    let r : String = foo(param)
                return r
}
func foo(_ par : String) -> String {
    let ret = bar(with: par)
                return ret
}
bar(with: "ahoj")
```

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

`func` *id* **`(`** *seznam_parametrů* **`) -> `** *návratový_typ* **`{`**
sekvence_příkazů
`}`

[6] Tzv. přetěžování funkcí (angl. *overloading*) je v IFJ23 součástí rozšíření OVERLOAD.

- Hlavička definice funkce sahá od klíčového slova **`func`** až po určení návratového typu funkce, pak následuje tělo funkce tvořené sekvencí příkazů (viz sekce 4.4) ve složených závorkách.
- Seznam parametrů je tvořen posloupností definic parametrů oddělených čárkou, přičemž za posledním parametrem se čárka neuvádí. Seznam může být i prázdný. Každá definice parametru obsahuje jméno, identifikátor a typ parametru:

  jméno_parametru identifikátor_parametru : *typ*

  Jméno parametru je použito při volání pro určení parametru (spolu s jeho správným pořadím), např. pro připomenutí významu parametru, a musí se lišit od identifikátoru parametru. Při použití _ se jméno parametru při volání vynechává a zapisuje se pouze term parametru. Identifikátor parametru slouží jako identifikátor v těle funkce pro získání hodnoty tohoto parametru. Při použití _ se parametr v těle funkce nepoužívá. Parametry jsou vždy předávány hodnotou a uvnitř těla funkce je nelze modifikovat. Příklad:

```
func concat(_ x : String, with y : String) -> String {
        let x = x + y
        return x + " " + y
}
let a = "ahoj "
var ct : String
ct = concat(a, with: "svete")
write(ct, a)
```

- V těle funkce jsou její parametry chápány jako předdefinované lokální proměnné s implicitní hodnotou danou skutečnými parametry, které nelze v těle funkce měnit. Výsledek funkce je dán provedeným příkazem návratu z funkce (viz sekce 4.4).

Každá funkce s návratovou hodnotou vrací hodnotu výrazu z příkazu **`return`**, avšak v případě chybějící návratové hodnoty kvůli neprovedení žádného příkazu **`return`**, nebo provedením příkazu **`return`**, ale s výrazem špatného typu (typ návratové hodnoty neodpovídá návratovému typu funkce), dochází k chybě 4.

Funkce bez návratové hodnoty (návratový typ je v hlavičce vynechán včetně **`->`**, tzv. *void-funkce*) je ukončena po posledním příkazu těla funkce nebo provedením příkazu **`return`** bez výrazu pro návratovou hodnotu. Definice vnořených funkcí není třeba podporovat.

4.3 Hlavní tělo programu

Hlavní tělo programu je neoznačená sekvence příkazů jazyka IFJ23 (kromě nepovoleného příkazu návratu z funkce), která se prolíná s definicemi funkcí (na nejvyšší úrovni příkazů, definice funkce tudíž nemůže narušit integritu příkazu, a to ani složeného[7]). Celá sekvence je ukončena koncem zdrojového souboru. Hlavní tělo programu může být tvořeno i prázdnou sekvencí příkazů, kdy je pouze vrácena návratová hodnota programu (možné hodnoty viz kapitola 2). Struktura jednotlivých příkazů je popsána v následující sekci.

[7] Složený příkaz je sekvence příkazů (i prázdná) uzavřená ve složených závorkách.

4.4 Syntaxe a sémantika příkazů

Dílčím příkazem se rozumí:

- *Příkaz definice proměnné*:
  `let` *id* **`:`** *typ* **`=`** *výraz*
  `var` *id* **`:`** *typ* **`=`** *výraz*
  Sémantika příkazu je následující: Příkaz nově definuje globální/lokální proměnnou, která v daném bloku ještě nebyla definována (lze takto překrýt proměnnou definovanou v případném obklopujícím bloku), jinak dojde k chybě 3. Příkaz provádí vyhodnocení výrazu *výraz* (viz kapitola 5) a případné přiřazení jeho hodnoty do levého operandu *id*. Levý operand musí být pro klíčové slovo **`var`** proměnná (tzv. l-hodnota) nebo pro klíčové slovo **`let`** nemodifikovatelná proměnná. Hodnotu nemodifikovatelné proměnné nelze po inicializaci již dále modifikovat. Je-li možné při překladu odvodit typ výrazu přiřazovaného do proměnné, tak je možné část '**`: typ`**' vypustit a typ proměnné *id* odvodit. Chybí-li *výraz* i *typ*, jedná se o syntaktickou chybu. Nelze-li chybějící typ odvodit (např. z hodnoty **`nil`**), dojde k chybě 8. Není-li typ zapsaného *výraz*u kompatibilní s uvedeným typem *typ*, nastane chyba 7.
  Pokud je typ proměnné/konstanty zadán, lze část '**`= výraz`**' vynechat.
- *Příkaz přiřazení*:
  id **`=`** *výraz*
  Sémantika příkazu je následující: Příkaz provádí vyhodnocení výrazu *výraz* (viz kapitola 5) a přiřazení jeho hodnoty do levého operandu *id*, který je modifikovatelnou proměnnou (tj. definovanou pomocí klíčového slova **`var`**). Pokud není hodnota výrazu typově kompatibilní s typem proměnné *id*, dojde k chybě 7.
- *Podmíněný příkaz*:

  `if` *výraz*
  `{`
  sekvence_příkazů$_1$
  `}`
  `else`
  `{`
  sekvence_příkazů$_2$
  `}`

  Sémantika příkazu je následující: Nejprve se vyhodnotí daný výraz pravdivostního typu. Pokud je vyhodnocený výraz pravdivý, vykoná se *sekvence_příkazů*$_1$, jinak se vykoná *sekvence_příkazů*$_2$. Pokud výsledná hodnota výrazu není pravdivostní (tj. pravda či nepravda - v základním zadání pouze jako výsledek aplikace relačních operátorů dle sekce 5.1), tak dojde k chybě 7.
  Místo pravdivostního výrazu *výraz* lze alternativně použít syntaxi: '**`let id`**', kde *id* zastupuje dříve definovanou (nemodifikovatelnou) proměnnou. Je-li pak proměnná *id* hodnoty **`nil`**, vykoná se *sekvence_příkazů*$_2$, jinak se vykoná *sekvence_příkazů*$_1$, kde navíc bude typ *id* upraven tak, že nebude (pouze v této sekvenci příkazů) zahrnovat hodnotu **`nil`** (tj. např. proměnná původního typu **`String?`** bude v tomto bloku typu

String). Rozšířenou variantu tohoto zápisu, kdy lze definovat zcela nové proměnné, nemusíte podporovat.

- *Příkaz cyklu*:

  while *výraz*
  {
  sekvence_příkazů
  }

  Příkaz cyklu se skládá z hlavičky a těla tvořeného *sekvencí_příkazů*.
  Sémantika příkazu cyklu je následující: Opakuje provádění sekvence dílčích příkazů (může být i prázdná) tak dlouho, dokud je hodnota výrazu pravdivá. Pravidla pro určení pravdivosti výrazu jsou stejná jako u výrazu v podmíněném příkazu.

- *Volání vestavěné či uživatelem definované funkce*:

  id **=** *název_funkce* **(***seznam_vstupních_parametrů***)**

  Seznam_vstupních_parametrů je seznam termů (viz sekce 3.1) včetně případného pojmenování a dvojtečkou oddělených čárkami[8]. Pojmenování parametru s dvojtečkou musí být uvedeno, pokud se nepoužije jméno _. Seznam může být i prázdný. Sémantika vestavěných funkcí bude popsána v kapitole 6. Sémantika volání uživatelem definovaných funkcí je následující: Příkaz zajistí předání parametrů hodnotou a předání řízení do těla funkce. V případě, že příkaz volání funkce obsahuje jiný počet nebo typy skutečných parametrů, než funkce očekává (tedy než je uvedeno v její hlavičce, a to i u vestavěných funkcí, včetně potenciálního předání **nil**, kde to není očekáváno), jedná se o chybu 4. Po dokončení provádění zavolané funkce je přiřazena návratová hodnota do proměnné *id* a běh programu pokračuje bezprostředně za příkazem volání právě provedené funkce. Je-li volána *void*-funkce, tak při pokusu o přiřazení výsledku do *id* dojde k chybě 7.
  V případě, že před *id* zapíšeme klíčové slovo **let** nebo **var** a případně za *id* volitelně určení typu, dojde před přiřazením hodnoty výsledku zavolané funkce nejprve k definici nové proměnné *id*.

- *Volání funkce bez navracení hodnoty (tj.* void-*funkce*):

  název_funkce **(***seznam_vstupních_parametrů***)**

  Při volání *void*-funkce jsou *seznam_vstupních_parametrů* a sémantika příkazu analogické předchozímu příkazu. Bez přiřazení návratové hodnoty lze volat i funkci, která dle své definice hodnotu vrací, a ta je poté zahozena.

- *Příkaz návratu z funkce*:

  return *výraz*

  Příkaz může být použit v těle libovolné funkce, ale ne v hlavním těle programu. Jeho sémantika je následující: Dojde k vyhodnocení výrazu *výraz* (tj. získání návratové hodnoty), okamžitému ukončení provádění těla funkce a návratu do místa volání, kam funkce vrátí vypočtenou návratovou hodnotu. V případě těla *void*-funkce musí být výraz vynechán, jinak nastane chyba 6. Uživatelem definovaná *void*-funkce nemusí příkaz **return** vůbec obsahovat. Chybí-li ve funkci vracející hodnotu výraz *výraz* u příkazu **return**, vede to na chybu 6. Pokud funkce s návratovou hodnotou vrací

[8] Parametrem volání funkce není výraz. Jedná se o součást nepovinného bodovaného rozšíření projektu FUNEXP.

návratovou hodnotu neočekávaného typu dle její definice, dochází k chybě 4.

5 Výrazy

Výrazy jsou tvořeny termy, závorkami a aritmetickými, řetězcovými a relačními operátory.

V IFJ23 jsou v rámci sémantické analýzy vstupního programu prováděny silné typové kontroly (tj. bez povolení implicitních typových kontrol). Ve výrazech si operátory kontrolují kompatibilitu operandů z hlediska jejich typů a případně i typu výsledku.

5.1 Aritmetické, řetězcové a relační operátory

Standardní binární operátory `+`, `-`, `*` značí sčítání, odčítání[9] a násobení. Jsou-li oba operandy typu `Int`, je i výsledek typu `Int`. Je-li jeden či oba operandy typu `Double`, výsledek je typu `Double`. Operátor `/` značí dělení dvou číselných operandů a výsledek je typu `Double`[10]. Pro provedení explicitního přetypování z `Double` na `Int` lze použít vestavěnou funkci `Double2Int`, naopak pak `Int2Double` (viz kapitola 6).

Řetězcový operátor `+` provádí se dvěma operandy typu `String` jejich konkatenaci.

Je-li hodnota levého operandu binárního operátoru `??` různá od `nil` je výsledkem tato hodnota, jinak je výsledkem hodnota pravého operandu, jehož typ musí odpovídat typu levého operandu bez zahrnutí `nil`. Postfixový unární operátor `!` slouží pro úpravu typu operandu tak, že na zodpovědnost programátora ve výsledku nemůže být `nil` (jinak může dojít k běhové chybě).

Pro operátor `==` platí: Pokud je první operand jiného typu než druhý operand, dochází k chybě 7. Číselné literály jsou při sémantických kontrolách implicitně přetypovány na potřebný číselný typ (tj. z `Int` na `Double`). Pokud jsou operandy stejného typu, tak se porovnají hodnoty daných operandů. Operátor `!=` je negací operátoru `==`.

Pro relační operátory `<`, `>`, `<=`, `>=` platí: Sémantika operátorů odpovídá jazyku Swift. Nelze při porovnání mít jeden z operandů jiného typu nebo druhý (např. jeden celé a druhý desetinné číslo), ani potenciálně obsahující `nil` (tj. případný výraz je nejprve potřeba převést na výraz typu bez hodnoty `nil`). U řetězců se porovnání provádí lexikograficky. Všechny typové nekompatibility ve výrazech jsou v IFJ23 hlášeny jako sémantická chyba.

Bez rozšíření BOOLTHEN není s výsledkem porovnání (pravdivostní hodnota) možné dále pracovat a lze jej využít pouze u podmínek příkazů `if` a `while`.

5.2 Priorita operátorů

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

Priorita	Operátory	Asociativita
1	`*` `/`	levá
2	`+` `-`	levá
3	`==` `!=` `<` `>` `<=` `>=`	bez asoc.
4	`??`	pravá

[9] Číselné literály jsou sice nezáporné, ale výsledek výrazu přiřazený do proměnné již záporný být může.
[10] Celočíselné dělení tudíž neuvažujte.

6 Vestavěné funkce

Překladač bude poskytovat některé základní vestavěné funkce, které bude možné využít v programech jazyka IFJ23. Pro generování kódu vestavěných funkcí lze výhodně využít specializovaných instrukcí jazyka IFJcode23.

Při použití špatného typu termu v parametrech následujících vestavěných funkcí dochází k chybě 4.

Vestavěné funkce pro načítání literálů a výpis termů:

- *Příkazy pro načítání hodnot*:
  `func readString() -> String?`
  `func readInt() -> Int?`
  `func readDouble() -> Double?`

  Vestavěné funkce ze standardního vstupu načtou jeden řádek ukončený odřádkováním nebo koncem souboru (EOF). Funkce **`readString`** tento řetězec vrátí bez symbolu konce řádku (načítaný řetězec nepodporuje escape sekvence). V případě **`readInt`** a **`readDouble`** jsou okolní bílé znaky ignorovány. Jakýkoli jiný nevhodný znak před či za samotným číslem je známkou špatného formátu a vede na návratovou hodnotu **`nil`**. Funkce **`readInt`** načítá a vrací celé číslo, **`readDouble`** desetinné číslo. V případě chybějící hodnoty na vstupu (např. načtení EOF) nebo jejího špatného formátu je vrácena hodnota **`nil`**.
- *Příkaz pro výpis hodnot*:
  `func write (` $term_1$ **`,`** $term_2$ **`, ..., `** $term_n$ **`)`**
  Vestavěný příkaz má libovolný počet parametrů tvořených termy oddělenými čárkou. Sémantika příkazu je následující: Postupně zleva doprava prochází termy (podrobněji popsány v sekci 3.1) a vypisuje jejich hodnoty na standardní výstup ihned za sebe bez žádných oddělovačů dle typu v patřičném formátu. Za posledním termem se též nic nevypisuje! Hodnota termu typu **`Int`** bude vytištěna pomocí `'%d'`[11], hodnota termu typu **`Double`** pak pomocí `'%a'`[12]. Hodnota **`nil`** je tištěna jako prázdný řetězec.
  Funkce **`write`** nemá návratovou hodnotu.

Vestavěné funkce pro konverzi číselných typů:

- **`func Int2Double(_`** *term* **`: Int) -> Double`** – Vrátí hodnotu celočíselného termu *term* konvertovanou na desetinné číslo. Pro konverzi z celého čísla využijte odpovídající instrukci z IFJcode23.
- **`func Double2Int(_`** *term* **`: Double) -> Int`** – Vrátí hodnotu desetinného termu *term* konvertovanou na celé číslo, a to oříznutím desetinné části. Pro konverzi z desetinného čísla využijte odpovídající instrukci z IFJcode23.

Vestavěné funkce pro práci s řetězci:

- **`func length(_`** *s* **`: String) -> Int`** – Vrátí délku (počet znaků) řetězce *s*. Např. **`length("x\nz")`** vrací 3.

[11] Formátovací řetězec standardní funkce **`printf`** jazyka C (standard C99 a novější).
[12] Formátovací řetězec **`printf`** jazyka C pro přesnou hexadecimální reprezentaci desetinného čísla.

- **`func substring(`**of s : **`String,`** startingAt i : **`Int,`** endingBefore j : **`Int) -> String?`** – 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 **`nil`**, nastane-li některý z těchto případů:
  - $i < 0$
  - $j < 0$
  - $i > j$
  - $i \geq$ **`length`**(s)
  - $j >$ **`length`**(s)
- **`func ord(_`** c : **`String) -> Int`** – Vrátí ordinální hodnotu (ASCII) prvního znaku v řetězci c. Je-li řetězec prázdný, vrací funkce **`0`**.
- **`func chr(_`** i : **`Int) -> String`** – Vrátí jednoznakový řetězec se znakem, jehož ASCII kód je zadán parametrem i. Hodnotu i mimo interval $[0; 255]$ řeší odpovídající instrukce IFJcode23.

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:

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`). Více viz sekce 12.2.

8 Příklady

Tato kapitola uvádí tři jednoduché příklady řídicích programů v jazyce IFJ23.

8.1 Výpočet faktoriálu (iterativně)

```
// Program 1: Vypocet faktorialu (iterativne)
/* Hlavni telo programu */
write("Zadejte cislo pro vypocet faktorialu\n")
let a : Int? = readInt()
if let a {
        if (a < 0) {
                write("Faktorial nelze spocitat\n")
        } else {
```

```
                var a = Int2Double(a)
                var vysl : Double = 1
                while (a > 0) {
                        vysl = vysl * a
                        a = a - 1
                }
                write("Vysledek je: ", vysl, "\n")
        }
} else {
        write("Chyba pri nacitani celeho cisla!\n")
}
```

8.2 Výpočet faktoriálu (rekurzivně)

```
// Program 2: Vypocet faktorialu (rekurzivne)

// Hlavni telo programu
write("Zadejte cislo pro vypocet faktorialu: ")
let inp : Int? = readInt()

// pomocna funkce pro dekrementaci celeho cisla o zadane cislo
func decrement(of n: Int, by m: Int) -> Int {
        return n - m
}

// Definice funkce pro vypocet hodnoty faktorialu
func factorial(_ n : Int) -> Int {
        var result : Int?
        if (n < 2) {
                result = 1
        } else {
                let decremented_n = decrement(of: n, by: 1)
                let temp_result = factorial(decremented_n)
                result = n * temp_result
        }
        return result!
}

// pokracovani hlavniho tela programu
if let inp {
        if (inp < 0)    { // Pokracovani hlavniho tela programu
                write("Faktorial nelze spocitat!")
        } else {
                let vysl = factorial(inp)
                write("Vysledek je: ", vysl)
        }
} else {
        write("Chyba pri nacitani celeho cisla!")
}
```

8.3 Práce s řetězci a vestavěnými funkcemi

```
/* Program 3: Prace s retezci a vestavenymi funkcemi */

var str1 = "Toto je nejaky text v programu jazyka IFJ23"
let str2 = str1 + ", ktery jeste trochu obohatime"
write(str1, "\n", str2, "\n")
let i = length(str1)
write("Pozice retezce \"text\" v str2: ", i, "\n")
write("Zadejte serazenou posloupnost vsech malych pismen a-h, ")

let newInput = readString()
if let newInput {
        str1 = newInput
        while (str1 != "abcdefgh") {
                write("Spatne zadana posloupnost, zkuste znovu:\n");
                str1 = readString() ?? ""
        }
}
```

9 Doporučení k testování

Programovací jazyk IFJ23 je schválně navržen tak, aby byl téměř kompatibilní s podmnožinou jazyka Swift 5[13]. Pokud si student není jistý, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ23, může si to ověřit následovně. Z Moodle si stáhne ze souborů k projektu soubor `ifj23.swift` obsahující kód, který doplňuje kompatibilitu IFJ23 s překladačem `swift` jazyka Swift 5.8 na serveru `merlin`. Soubor `ifj23.swift` obsahuje definice vestavěných funkcí, které jsou součástí jazyka IFJ23, ale chybí v potřebné formě v jazyce Swift 5.8, nebo tyto redefinují.

Váš program v jazyce IFJ23 uložený například v souboru `testPrg.swift` pak lze provést na serveru `merlin` například pomocí příkazu:

```
swift <(cat testPrg.swift ifj23.swift) < test.in > test.out
```

Tím lze jednoduše zkontrolovat, co by měl provést zadaný zdrojový kód, resp. vygenerovaný cílový kód. Je ale potřeba si uvědomit, že jazyk Swift 5.8 je nadmnožinou jazyka IFJ23, a tudíž může zpracovat i konstrukce, které nejsou v IFJ23 povolené (např. bohatší syntaxe a sémantika většiny příkazů, či dokonce zpětné nekompatibility). Výčet těchto odlišností bude uveden v Moodle IFJ a můžete jej diskutovat na fóru předmětu IFJ.

10 Cílový jazyk IFJcode23

Cílový jazyk IFJcode23 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,

[13] Online dokumentace k Swift: `https://www.swift.org/documentation/`

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 IFJcode23 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode23 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode23
```

10.1 Hodnotící interpret `ic23int`

Pro hodnocení a testování mezikódu v IFJcode23 je k dispozici interpret pro příkazovou řádku (`ic23int`):

```
ic23int 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 IFJcode23.
- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode23.
- 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. IFJcode23 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 IFJcode23 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 IFJ22.

Zápis každé konstanty v IFJcode23 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). Např. `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 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 je 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 [0; 255], 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.

INT2FLOATS/FLOAT2INTS/ INT2CHARS/STRI2INTS 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} (včetně případné konverze vstupní hodnoty float při zadaném typu int) a uloží tuto hodnotu do proměnné ⟨*var*⟩. Formát hodnot je kompatibilní s chováním vestavěných funkcí **`readString`**, **`readInt`** a **`readDouble`** jazyka IFJ23.

WRITE ⟨*symb*⟩ Výpis hodnoty na standardní výstup
Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **`write`** jazyka IFJ23 včetně výpisu desetinných čísel pomocí formátovacího řetězce "%a".

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.

10.4.7 Instrukce pro řízení toku programu

Neterminál ⟨*label*⟩ označuje návěští, které slouží pro označení pozice v kódu IFJcode23. 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 **vedoucího** týmu. Archiv nesmí obsahovat adresářovou strukturu ani speciální či spustitelné soubory. Názvy všech souborů budou obsahovat pouze písmena[14], 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 archív bude povinně obsahovat soubor **`rozdeleni`**, ve kterém zohledníte dělení bodů mezi jednotlivé členy týmu (i při požadavku na rovnoměrné dělení). Na každém řádku je uveden login jednoho člena týmu, bez mezery je následován dvojtečkou a po ní je bez mezery uveden požadovaný celočíselný počet procent bodů bez uvedení znaku %. Každý řádek (i poslední) je poté ihned ukončen jedním znakem ⟨LF⟩ (ASCII hodnota 10, tj. unixové ukončení řádku, ne windowsovské!). Obsah souboru bude vypadat například takto (Ⓛ zastupuje unixové odřádkování):

```
xnovak01:30Ⓛ
xnovak02:40Ⓛ
xnovak03:30Ⓛ
xnovak04:00Ⓛ
```

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 kapitola i několik rad pro zdárné řešení tohoto projektu a výčet rozšíření za prémiové body.

12.1 Závazné metody pro implementaci překladače

Projekt bude hodnocen pouze jako funkční celek, a nikoli jako soubor separátních, společně nekooperujících modulů. Při tvorbě lexikální analýzy využijete znalosti konečných automatů. Při konstrukci syntaktické analýzy založené na LL-gramatice (vše kromě

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

výrazů) **povinně** využijte buď **metodu rekurzivního sestupu** (doporučeno), nebo prediktivní analýzu řízenou LL-tabulkou. Výrazy zpracujte pouze pomocí **precedenční syntaktické analýzy**. Vše bude probíráno na přednáškách v rámci předmětu IFJ. Implementace bude provedena **v jazyce C**, čímž úmyslně omezujeme možnosti použití objektově orientované implementace. Návrh implementace překladače je zcela v režii řešitelských týmů. Není dovoleno spouštět další procesy a vytvářet nové či modifikovat existující soubory (ani v adresáři `/tmp`). Nedodržení těchto metod bude penalizováno značnou ztrátou bodů!

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[15] nebo diagramy, které nejsou vaše původní (kopie z přednášek, sítě, WWW, …).

[15] Vyjma obyčejného loga fakulty na úvodní straně.

- 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 lex/flex, yacc/-bison či jiných podobného ražení 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 archívu nepřikládejte!

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

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, kompilát interpretovat naším interpretem jazyka IFJcode22 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

Při řešení je pochopitelně možné využít vlastní výpočetní techniku. 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 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.

Maximální počet bodů získatelný na jednu osobu za programovou implementaci je **20** 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 studentů 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 a studentům zaslána 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.

12.7 Registrovaná rozšíření

V případě implementace některých registrovaných rozšíření bude odevzdaný archív 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 20 bodů.

12.7.1 Bodové hodnocení některých rozšíření jazyka IFJ23

Popis rozšíření vždy začíná jeho identifikátorem. Většina těchto rozšíření je založena na dalších vlastnostech jazyka Swift 5. Podrobnější informace lze získat ze specifikace jazyka[2] Swift 5. Do dokumentace je potřeba (kromě zkratky na úvodní stranu) také uvést, jak jsou implementovaná rozšíření řešena.

- TODO: OVERLOAD – podpora redefinování funkcí stejného jména, ale dostatečně odlišné signatury v parametrech.
- INTERPOLATION – podpora použití výrazů včetně případného jednoho volání funkce přímo v zápisu řetězců pomocí párových závorek **`\(`** a **`)`** a náhradě toho zápisu při práci s takovým řetězcem.[16]
- BOOLTHEN: Podpora typu **`Bool`**, booleovských hodnot **`true`** a **`false`**, booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**`!`**, **`&&`** a **`||`** včetně zkratového vyhodnocení), jejichž priorita a asociativita odpovídá jazyku Swift[2]. Pravdivostní hodnoty lze porovnávat jen operátory **`==`** a **`!=`**. Podporujte výpisy hodnot typu **`Bool`** a přiřazování výsledku booleovského výrazu do proměnné. Dále podporujte rozšířený podmíněný příkaz **`if-else if-else`** dle manuálu. Části **`else if`** a **`else`** jsou tedy volitelné (+1,0 bodu).
- CYCLES: Překladač bude podporovat i cyklus **`for-in`**:
  `for` *jméno_proměnné* **`in`** *rozsah* **`{`**
  sekvence_příkazů
  `}` Sémantika příkazu viz manuál[2]. *Jméno_proměnné* může být nahrazeno _, pokud se v sekvenci příkazů nepoužívá. Rozsahem může být Closed Range Operator (*a*`...`*b*) nebo Half-Open Range Operator (*a*`..<`*b*), kde *a* a *b* jsou výrazy typu **`Int`**. Podporujte i příkazy **`break`** a **`continue`** (+1,0 bodu). Příklad:

```
let hours = 1
for time in 0..<hours*60 {
```

[16] https://docs.swift.org/swift-book/documentation/the-swift-programming-language/stringsandcharacters#String-Interpolation

```
        if (time == 30) {
            continue
        }
        write(time, "\n")
    }
```

- FUNEXP: Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce (+1,5 bodu).

13 Opravy zadání

- 23. 9. 2023 – Oprava několika překlepů, nejasných formulací a pravopisných chyb, doplnění popisu rozšíření INTERPOLATION.