ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL

Zbyněk Křivka, Adam Kövári, Dominika Regéciová
email: {krivka, ikovari, iregeciova}@fit.vut.cz
22. září 2021

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ21.
Informace: diskuzní fórum a wiki stránky předmětu IFJ v IS FIT.
Pokusné odevzdání: čtvrtek 25. listopadu 2021, 23:59 (nepovinné).
Datum odevzdání: středa 8. prosince 2021, 23:59.
Způsob odevzdání: prostřednictvím IS FIT do datového skladu předmětu IFJ.

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 - Prémiové body“ v IFJ.

Řešitelské týmy:

- Projekt budou řešit tří až čtyřčlenné týmy. Týmy s jiným počtem členů jsou nepřípustné.
- Registrace do týmů se provádí přihlášením na příslušnou variantu zadání v IS FIT. Registrace je dvoufázová. V první fázi se na jednotlivé varianty projektu přihlašují **pouze** vedoucí týmů (kapacita je omezena na 1). Ve druhé fázi se pak sami doregistrují ostatní členové (kapacita bude zvýšena na 4). Vedoucí týmů budou mít plnou

pravomoc nad složením svého týmu. Rovněž vzájemná komunikace mezi vyučujícími a týmy bude probíhat nejlépe prostřednictvím vedoucích (ideálně v kopii dalším členům týmu). Ve výsledku bude u každého týmu prvně zaregistrovaný člen považován za vedoucího tohoto týmu. Všechny termíny k projektu najdete v IS FIT a další informace na stránkách předmětu[1].

- Zadání obsahuje dvě varianty, které se liší pouze ve způsobu implementace tabulky symbolů a jsou identifikované římskou číslicí I nebo II. Každý tým má své identifikační číslo, na které se váže vybraná varianta zadání. Výběr variant se provádí přihlášením do skupiny daného týmu v IS FIT.

2 Zadání

Vytvořte program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ21 a přeloží jej do cílového jazyka IFJcode21 (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).
- 3 - sémantická chyba v programu – nedefinovaná funkce/proměnná, pokus o redefinici proměnné, atp.
- 4 - sémantická chyba v příkazu přiřazení (typová nekompatibilita).
- 5 - sémantická chyba v programu – špatný počet/typ parametrů či návratových hodnot u volání funkce či návratu z funkce.
- 6 - sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.
- 7 - ostatní sémantické chyby.
- 8 - běhová chyba při práci s neočekávanou hodnotou **`nil`**.
- 9 - běhová chyba celočíselného dělení nulovou konstantou.
- 99 - interní chyba překladače tj. neovlivněná vstupním programem (např. chyba alokace paměti atd.).

Překladač bude načítat řídicí program v jazyce IFJ21 ze standardního vstupu a generovat výsledný mezikód v jazyce IFJcode21 (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 IFJcode21 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 IFJ21 je zjednodušenou podmnožinou jazyka Teal[2], který vznikl doplněním statického typování do jazyka Lua[3]. Teal je staticky typovaný[4] imperativní jazyk.

3.1 Obecné vlastnosti a datové typy

V programovacím jazyce IFJ21 **záleží** na velikosti písmen u identifikátorů i klíčových slov (tzv. *case-sensitive*).

- *Identifikátor* je definován jako neprázdná posloupnost písmen, číslic a znaku podtržítka ('_') začínající písmenem nebo podtržítkem[5].
- Jazyk IFJ21 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam, a proto se nesmějí vyskytovat jako identifikátory[6]:

do,	**else**,	**end**,	**function**,
global,	**if**,	**local**,	**nil**,
require,	**return**,	**then**,	**while**.

- *Celočíselný literál* (rozsah C-int) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě[7].
- *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)[8].
- *Řetězcový literál* je oboustranně ohraničen dvojitými uvozovkami (**"**, ASCII hodnota 34). Tvoří jej libovolný počet znaků zapsaných na jediném řádku programu. 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**', '**\t**', '****'. Jejich význam se shoduje s odpovídajícími znakovými konstantami jazyka Lua[9]. Neexistující escape sekvence vede na lexikální chybu. Znak v řetězci může být zadán

[2] `https://teal-language.org/`; na serveru Merlin je pro studenty k dispozici překladač `tl` verze 0.13.2.

[3] `http://www.lua.org/`; skriptovací jazyk často používaný například při vývoji her.

[4] Jednotlivé proměnné/funkce mají datový typ určen deklarací/definicí nebo staticky odvozen na základě zdrojového kódu.

[5] Na rozdíl od jazyka IFJ21, jazyk Teal u identifikátorů začínajících podtržítkem nehlásí varování (např. při redefinici proměnné nebo nevyužití proměnné).

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

[7] Přebytečná počáteční číslice `0` je ignorována.

[8] Přebytečné počáteční číslice `0` v celočíselné části či exponentu jsou ignorovány.

[9] `http://www.lua.org/manual/5.3/manual.html#3.1`

také pomocí obecné escape sekvence '*****ddd*', kde *ddd* je právě třímístné desítkové číslo od **001** do **255**.
Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

```
"Ahoj\n\"Sve'te \\\034"
```

reprezentuje řetězec
Ahoj
"Sve'te \". Neuvažujte řetězce, které obsahují vícebajtové znaky kódování Unicode (např. UTF-8).

- *Datové typy* pro jednotlivé literály jsou označeny **integer**, **number** a **string**, kde do typu **number** spadají i celočíselné hodnoty. Speciálním případem je typ *nil*, který nabývá pouze hodnoty **nil**. Hodnoty **nil** mohou nabývat všechny proměnné.
- *Term* je libovolný literál (celočíselný, desetinný, řetězcový či **nil**) nebo identifikátor proměnné.
- Jazyk IFJ21 podporuje *řádkové* i *blokové* komentáře stejně jako jazyk Teal/Lua. Řádkový komentář začíná dvojicí pomlček ('**--**', ASCII hodnota 45) a za komentář je považováno vše, co následuje až do konce řádku. Blokový komentář začíná posloupností symbolů '**--[[**' a je ukončen dvojicí symbolů '**]]**'. Vnořené blokové komentáře neuvažujte.

4 Struktura jazyka

IFJ21 je strukturovaný programovací jazyk podporující definice proměnných a uživatelských funkcí včetně jejich rekurzivního volání. Vstupním bodem prováděného programu je neoznačená nesouvislá sekvence příkazů volání funkce bez navracení hodnoty mezi deklaracemi a definicemi uživatelských funkcí, tzv. *hlavní tělo* programu.

4.1 Základní struktura jazyka

Program se skládá z prologu následovaného sekvencí deklarací a definic uživatelských funkcí a příkazů volání funkce bez navracení hodnoty. V tělech uživatelských funkcí lze potom definovat lokální proměnné, používat příkazy přiřazení, větvení, iterace a volání funkcí.

Prolog[10] se skládá z jednoho řádku:

```
require "ifj21"
```

Mezi jednotlivými tokeny se může vyskytovat libovolný počet bílých znaků (mezera, tabulátor, komentář a odřádkování), takže jednotlivé konstrukce jazyka IFJ21 lze zapisovat na jednom či více řádcích nebo může být na jednom řádku více příkazů jazyka IFJ21. Je-li to třeba, jsou příkazy odděleny bílými znaky[11]. Na začátku a konci zdrojového textu se smí vyskytovat libovolný počet bílých znaků.

[10] Prolog mohou prokládat komentáře a prázdné řádky a slouží především kvůli kompatibilitě s programy jazyka Teal.
[11] Na rozdíl od jazyka Teal, kde je možné použít i středník.

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

4.1.1 Proměnné

Proměnné jazyka IFJ21 jsou pouze lokální a mají rozsah platnosti v bloku, ve kterém byly definovány, od místa jejich definice až po konec tohoto bloku. *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 definicí stejně pojmenované proměnné.

Definice proměnné se provádí pomocí příkazu definice proměnné s povinným určením typu a s volitelnou inicializací. Detailní syntaxe bude popsána v sekci 4.3.

V rámci jednoho bloku nelze definovat více proměnných stejného jména. Dále nelze definovat proměnnou stejného jména jako některá již deklarovaná či definovaná funkce.

Není-li typ inicializačního výrazu staticky (při překladu) kompatibilní s typem inicializované proměnné, jde o chybu 4. Výraz, jehož typ lze odvodit staticky, může obsahovat literály, již definované proměnné, operátory i závorky, nebo se může jednat o volání funkce s návratovou hodnotou kompatibilního typu. Definice proměnné stejného jména jako má jiná proměnná ve stejném bloku vede na chybu 3. Každá proměnná musí být definována před jejím použitím, jinak se jedná o sémantickou chybu 3. Při definici proměnné stejného jména jako některá proměnná z nadřazené úrovně je viditelná pouze později definovaná proměnná, která je na daném místě platná. Ostatní proměnné stejného jména mohou být platné, i když nejsou v dané části programu viditelné.

4.2 Deklarace a definice uživatelských funkcí

Definice funkce se skládá z hlavičky a těla funkce. Každá uživatelská funkce s daným identifikátorem je definována nejvýše jednou[12], jinak dochází k chybě 3. Definice funkce nemusí vždy lexikálně předcházet kódu pro volání této funkce. Uvažujte například vzájemné rekurzivní volání funkcí (tj. funkce f volá funkci g, která opět může volat funkci f).

V případě, že je volána funkce, která ještě nebyla definována, musí jejímu volání předcházet alespoň její deklarace. Funkce může být deklarována nejvýše jednou, jinak dochází k chybě 3. K té také dochází, pokud u deklarace a definice funkce neodpovídají seznamy parametrů nebo seznamy návratových typů, či pokud deklarovaná funkce není nakonec definována. Příklad:

```
global foo : function(string) : string

function bar(param : string) : string
    return foo (param)
end

function foo(param:string):string
    return bar(param)
end
```

[12]Tzv. přetěžování funkcí (angl. *overloading*) není v IFJ21 podporováno.

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

function *id* **(** *seznam_parametrů* **) :** *seznam_návratových_typů*
složený_příkaz
end

- Hlavička definice funkce sahá od klíčového slova **function** až po určení návratového typu funkce, pak následuje tělo funkce zakončené klíčovým slovem **end**.
- 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 identifikátor parametru a za dvojtečkou jeho datový typ:
  identifikátor_parametru **:** *typ*
  Parametry jsou vždy předávány hodnotou.
- Seznam návratových typů je posloupnost datových typů, jejichž hodnoty funkce vrací. Oddělovačem v seznamu návratových typů je opět čárka. Příklad:

```
require "ifj21"
function concat (x : string, y : string) : string, integer
  return x .. y, 0  end
function main()
  local x : string
  local ret: integer
  x, ret = concat("ahoj", "svete")
end
main()
```

  Seznam návratových typů může být i prázdný, což je využito u funkcí bez navracení hodnoty, které mohou být volány např. v hlavním těle programu IFJ21.
- Tělo funkce je tvořeno sekvencí příkazů (viz sekce 4.3) ukončených odpovídajícím klíčovým slovem **end**. Parametry funkce jsou chápány jako předdefinované lokální proměnné.

Každá funkce s návratovou hodnotou vrací jednu nebo více hodnot dané vyhodnocením výrazu v příkazu **return**. V případě chybějící návratové hodnoty kvůli neprovedení žádného příkazu **return** bude navracen odpovídající počet hodnot **nil**. Pokud typy návratových hodnot neodpovídají hlavičce funkce, dojde k chybě 5. Definice vnořených funkcí je zakázána.

4.3 Syntaxe a sémantika příkazů

Není-li řečeno jinak, tak každá *sekvence příkazů* v následujících popisech strukturovaných příkazů tvoří nový blok, který má vlastní rozsah platnosti proměnných v něm definovaných. Sekvence příkazů může být i prázdná.
Dílčím příkazem se rozumí:

- *Příkaz definice proměnné*:
  local *id* **:** *datovytyp* **=** *init_výraz_ci_volani_fce*

Sémantika příkazu je následující: Příkaz nově definuje lokální proměnnou, která v daném bloku ještě nebyla definována (lze takto překrýt proměnnou definovanou v případném obklopujícím bloku), jinak dojde k chybě 3. Typ nově definované proměnné *id* je tedy dán staticky jako *datovytyp* a nelze jej měnit. Je-li přítomen i volitelný inicializační výraz či inicializační volání funkce s návratovou hodnotou, je *id* inicializováno výslednou hodnotou, jinak je implicitně inicializováno na **`nil`**. Vyhodnocení inicializačního výrazu či inicializačního volání funkce je popsáno v příkazu přiřazení a příkazu volání funkce.

- *Příkaz přiřazení*:
  id_1 **`,`** id_2 **`, ...,`** id_n **`=`** *výraz*$_1$ **`,`** *výraz*$_2$ **`, ...,`** *výraz*$_n$
  kde $n \geq 1$. Syntakticky slouží čárka (’**`,`**’) jako oddělovač při $n \geq 2$, takže za posledním identifikátorem či výrazem se čárka nepíše. Sémantika příkazu je následující: Příkaz provádí přiřazení hodnot operandů vpravo (výrazy *výraz*$_1$ až *výraz*$_n$; viz kapitola 5) po řadě do odpovídajících proměnných id_1 až id_n tak, že nejprve provede vyhodnocení všech výrazů (operandů napravo v pořadí zprava do leva), zapamatování si těchto výsledků a následně přiřazení těchto výsledků jednotlivým proměnným. Všechny proměnné nalevo od **`=`** musí být dříve definované a platné.
- *Podmíněný příkaz*:
  `if` *výraz* **`then`**
  sekvence_příkazů$_1$
  `else`
  sekvence_příkazů$_2$
  `end`

  Sémantika příkazu je následující: Nejprve se vyhodnotí daný výraz. 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 se hodnota **`nil`** bere jako nepravda a ostatní hodnoty jako pravda (včetně prázdného řetězce nebo nuly).
- *Příkaz cyklu*:
  `while` *výraz* **`do`**
  sekvence_příkazů
  `end`
  Příkaz cyklu se skládá z hlavičky a těla tvořeného *sekvencí_příkazů*.
  Sémantika příkazu cyklu je následující: Opakuje se provádění *sekvence_příkazů* (viz příkazy v této sekci) tak dlouho, dokud je hodnota výrazu pravdivá. Pravidla pro určení pravdivosti výrazu jsou stejná jako u výrazu v podmíněném příkazu. Proměnné definované až v těle cyklu nejsou viditelné v hlavičce cyklu.
- *Volání vestavěné či uživatelem definované funkce*:
  id_1, id_2, ..., id_n **`=`** *název_funkce* **`(`***seznam_vstupních_parametrů***`)`**
  kde $n \geq 1$. Syntakticky slouží čárka (’**`,`**’) jako oddělovač při $n \geq 2$, takže za posledním skutečným parametrem se čárka nepíše.
  Seznam_vstupních_parametrů je seznam termů (viz sekce 3.1) oddělených čárka-

mi[13]. 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 parametrů, než funkce očekává (tedy než je uvedeno v její hlavičce, a to i u vestavěných funkcí), jedná se o chybu 5. Po dokončení provádění zavolané funkce je přiřazena návratová hodnota (hodnoty) do proměnné/proměnných id_1 až id_n a běh programu pokračuje bezprostředně za příkazem volání právě provedené funkce. Neobsahuje-li tělo funkce příkaz **`return`**, vrací funkce odpovídající počet hodnot **`nil`**. Pokud funkce vrací méně hodnot, než je očekáváno dle počtu proměnných id_1 až id_n, dojde k chybě 5. Pokud naopak funkce vrací více hodnot (tj. více než n), jsou tyto hodnoty zahazovány. Typová nekompatibilita mezi návratovou hodnotou a odpovídající proměnnou pro její uložení vede na chybu 5. Všechny proměnné nalevo od **`=`** již musí být definovány.

- *Volání funkce bez navracení hodnoty*:
  název_funkce **`(`***seznam_vstupních_parametrů***`)`**
  Při volání funkce bez navracení hodnoty je sémantika příkazu analogická předchozímu, ale případné návratové hodnoty nebudou využity. *Seznam_vstupních_parametrů* má stejný význam jako v předchozí řídicí struktuře. Tímto způsobem lze volat i funkci, která dle své definice nějaké hodnoty vrací, ty jsou však při takovémto volání zahozeny. Tento druh volání se smí vyskytovat jako jediný v hlavním těle programu (tj. mezi definicemi/deklaracemi funkcí), kdy je typické, že je takto na konci programu volána hlavní funkce nebo více funkcí.
- *Příkaz návratu z funkce*:
  `return` *seznam_výrazů*
  Příkaz je typicky použit v těle funkce, která má neprázdný návratový typ. U funkcí bez návratové hodnoty může být příkaz návratu z funkce zcela vynechán nebo použita varianta bez seznamu výrazů. Jeho sémantika je následující: Dojde k vyhodnocení jednotlivých čárkou oddělených výrazů v *seznam_výrazů* (tj. získání n-tice návratových hodnot, kde $n \geq 0$, a to zleva doprava), okamžitému ukončení provádění těla funkce a návratu do místa volání, kam funkce vrátí vypočtenou návratovou hodnotu/hodnoty. Je-li počet výrazů výsledných hodnot nekompatibilní s návratovými typy dané funkce, jsou chybějící hodnoty doplněny speciální hodnotou **`nil`** a přebývající způsobí chybu 5.

5 Výrazy

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

V IFJ21 je typ **`integer`** podtypem typu **`number`**, takže je možné využít typ **`integer`** tam, kde se očekává typ **`number`** (tzv. typová kompatibilita). Je-li třeba, tak dojde i k implicitní typové konverzi z **`integer`** na **`number`**.

Pro chybné kombinace datových typů ve výrazech vracejte chybu 6.

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

5.1 Aritmetické, řetězcové a relační operátory

Standardní binární operátory `+`, `-`, `*` značí sčítání, odčítání[14] a násobení. Jsou-li oba operandy typu `integer`, je i výsledek typu `integer`. Jsou-li oba operandy typu `number` nebo jeden `integer` a druhý `number`, výsledek je typu `number`. Infixový binární operátor `..` provádí se dvěma řetězcovými operandy konkatenaci. Prefixový unární operátor `#` slouží pro získání délky (počet znaků) řetězce zadaného jako jediný operand vpravo. Např. `#"x\nz"` vrací 3.

Operátor `/` značí dělení dvou číselných operandů a výsledek je typu `number`. Jsou-li oba operandy celočíselné, lze využít operátor `//` pro celočíselné dělení. Při dělení nulou operátorem `//` ale i `/` nastává chyba 9. Pro provedení explicitního přetypování z `number` na `integer` lze použít vestavěnou funkci `tointeger` (viz kapitola 6).

Pro relační operátory `<`, `>`, `<=`, `>=`, `==`, `~=` platí, že výsledkem porovnání je pravdivostní hodnota a že mají stejnou sémantiku jako v jazyce Teal/Lua. Tyto operátory pracují s operandy stejných nebo kompatibilních typů, a to `integer`, `number` nebo `string`. Pomocí `==` a `~=` lze provádět i test na speciální hodnotu `nil`, ale v případě aplikace jiných operátorů na `nil` dochází k běhové chybě 8. U řetězců se porovnání provádí lexikograficky. Bez rozšíření BOOLTHEN není s výsledkem porovnání 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
0	`#`	unární
1	`*` `/` `//`	levá
2	`+` `-`	levá
3	`..`	pravá
4	`<` `<=` `>` `>=` `==` `~=`	levá[15]

6 Vestavěné funkce

Překladač bude poskytovat některé základní vestavěné funkce, které bude možné využít v programech jazyka IFJ21. Pro generování kódu vestavěných funkcí lze výhodně využít specializovaných instrukcí jazyka IFJcode21.

Při použití špatného typu termu v parametrech následujících vestavěných funkcí dochází k chybě 5.

Vestavěné funkce pro načítání literálů a výpis termů:

[14] Číselné literály jsou sice nezáporné, ale výsledek výrazu přiřazený do proměnné již záporný být může.

[15] Asociativitu relačních operátorů není v základu třeba implementovat, případně viz rozšíření BOOLTHEN.

- *Příkazy pro načítání hodnot*:
  function reads() : string
  function readi() : integer
  function readn() : number

  Vestavěné funkce ze standardního vstupu načtou jeden řádek ukončený odřádkováním. Funkce **reads** tento řetězec vrátí bez symbolu konce řádku (načítaný řetězec nepodporuje escape sekvence). V případě **readi** a **readn** 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 **readi** načítá a vrací celé číslo, **readn** desetinné číslo. Obě funkce podporují i načítání hexadecimálního zápisu čísla (např. `0x1FA3` nebo `0x1F.F1p-1`, kde je šestnáctková soustava detekována podřetězci `0x` a `p`). V případě chybějící hodnoty na vstupu (např. načtení EOF) nebo jejího špatného formátu je vrácena hodnota **nil**.
- *Příkaz pro výpis hodnot*:
  function 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 **integer** bude vytištěna pomocí `'%d'`[16], hodnota termu typu **number** pak pomocí `'%a'`[17]. Funkce **write** nemá návratovou hodnotu.

Vestavěné funkce pro konverzi číselných typů:

- **function tointeger(**f **: number) : integer** – Vrací hodnotu desetinného parametru *f* převedenou na celočíselnou hodnotu oříznutím desetinné části. Je-li *f* **nil**, vrací funkce též **nil**.

Vestavěné funkce pro práci s řetězci:

- **function substr(**s **: string,** i **: number,** j **: number) : string** – Vrátí podřetězec zadaného řetězce *s*. Druhým parametrem *i* je dán index začátku požadovaného podřetězce (počítáno od jedničky) a třetím parametrem *j* je index konce požadovaného podřetězce (počítáno od jedničky). Je-li index *i*, nebo *j* mimo meze 1 až **#**s, nebo je-li $j < i$, vrací funkce prázdný řetězec. Je-li některý parametr **nil**, nastává chyba 8.
- **function ord(**s **: string,** i **: integer) : integer** – Vrátí ordinální hodnotu (ASCII) znaku na pozici *i* v řetězci *s*. Je-li jeden z parametrů **nil**, nastává chyba 8. Je-li index *i* mimo meze řetězce (1 až **#**s), vrací funkce **nil**.
- **function chr(**i **: integer) : string** – Vrátí jednoznakový řetězec se znakem, jehož ASCII kód je zadán parametrem *i*. Případ, kdy je *i* mimo interval $[0; 255]$, vede na hodnotu **nil**. Je-li *i* **nil**, nastává chyba 8.

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

7 Implementace tabulky symbolů

Tabulka symbolů bude implementována pomocí abstraktní datové struktury, která je ve variantě zadání pro daný tým označena římskými číslicemi I-II, a to následovně:

I) Tabulku symbolů implementujte pomocí binárního vyhledávacího stromu.

II) Tabulku symbolů implementujte pomocí tabulky s rozptýlenými položkami.

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 IFJ21.

8.1 Výpočet faktoriálu (iterativně)

```
-- Program 1: Vypocet faktorialu (iterativne)
require "ifj21"

function main() -- uzivatelska funkce bez parametru
  local a : integer
  local vysl : integer = 0
  write("Zadejte cislo pro vypocet faktorialu\n")
  a = readi()
  if a == nil then
      write("a je nil\n") return
  else
  end
  if a < 0 then
    write("Faktorial nelze spocitat\n")
  else
    vysl = 1
    while a > 0 do
      vysl = vysl * a a = a - 1  -- dva prikazy
    end
    write("Vysledek je: ", vysl, "\n")
  end
end

main()  -- prikaz hlavniho tela programu
```

8.2 Výpočet faktoriálu (rekurzivně)

```
-- Program 2: Vypocet faktorialu (rekurzivne)
require "ifj21"

function factorial(n : integer) : integer
  local n1 : integer = n - 1
  if n < 2 then
    return 1
```

```
  else
    local tmp : integer = factorial(n1)
    return n * tmp
  end
end

function main()
  write("Zadejte cislo pro vypocet faktorialu: ")
  local a : integer = readi()
  if a ~= nil then
                if a < 0 then
                        write("Faktorial nejde spocitat!", "\n")
                else
                        local vysl : integer = factorial(a)
                        write("Vysledek je ", vysl, "\n")
    end
  else
     write("Chyba pri nacitani celeho cisla!\n")
  end
end

main()
```

8.3 Práce s řetězci a vestavěnými funkcemi

```
-- Program 3: Prace s retezci a vestavenymi funkcemi
require "ifj21"
function main()
  local s1 : string = "Toto je nejaky text"
  local s2 : string = s1 .. ", ktery jeste trochu obohatime"
  write(s1, "\010", s2)local s1len:integer=#s1 local s1len4: integer=s1len
  s1len = s1len - 4 s1 = substr(s2, s1len, s1len4) s1len = s1len + 1
  write("4 znaky od", s1len, ". znaku v \"", s2, "\":", s1, "\n")
  write("Zadejte serazenou posloupnost vsech malych pismen a-h, ")
        write("pricemz se pismena nesmeji v posloupnosti opakovat: ")
  s1 = reads()
  if s1 ~= nil then
    while s1 ~= "abcdefgh" do
      write("\n", "Spatne zadana posloupnost, zkuste znovu:")
      s1 = reads()
    end
  else
  end
end
main()
```

9 Doporučení k testování

Programovací jazyk IFJ21 je schválně navržen tak, aby byl téměř kompatibilní s podmnožinou jazyka Teal[18]. Pokud si student není jistý, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ21, může si to ověřit následovně. Z IS FIT si stáhne ze

[18] Online dokumentace k Teal: `https://teal-language.org/` a k výchozímu jazyku Lua: `http://www.lua.org/`

Souborů k předmětu IFJ ze složky *Projekt* soubor `ifj21.tl` obsahující kód, který doplňuje kompatibilitu IFJ21 s překladačem `tl` jazyka Teal 0.13.2 na serveru `merlin`. Soubor `ifj21.tl` obsahuje definice vestavěných funkcí, které jsou součástí jazyka IFJ21, ale chybí v potřebné formě v jazyce Teal/Lua.

Váš program v jazyce IFJ21 uložený například v souboru `testPrg.tl`[19] pak lze provést na serveru `merlin` například pomocí příkazu:

```
tl run testPrg.tl < 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 Teal je nadmnožinou jazyka IFJ21, a tudíž může zpracovat i konstrukce, které nejsou v IFJ21 povolené (např. bohatší syntaxe a sémantika většiny příkazů, či dokonce zpětné nekompatibility). Výčet těchto odlišností bude uveden na wiki stránkách a můžete jej diskutovat na fóru předmětu IFJ.

10 Cílový jazyk IFJcode21

Cílový jazyk IFJcode21 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 IFJcode21 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode21 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode21
```

10.1 Hodnotící interpret `ic21int`

Pro hodnocení a testování mezikódu v IFJcode21 je k dispozici interpret pro příkazovou řádku (`ic21int`):

```
ic21int 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.

[19] Používejte výhradně příponu `tl`, aby byl rozpoznán jazyk Teal se statickým typováním a ne Lua, kde jsou pouze kontroly za běhu.

- 51 - chyba při analýze (lexikální, syntaktická) vstupního kódu v IFJcode21.
- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode21.
- 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. IFJcode21 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 IFJcode21 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 IFJ21.

Zápis každé konstanty v IFJcode21 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 **`reads`**, **`readi`** a **`readn`** jazyka IFJ21.

WRITE ⟨*symb*⟩ Výpis hodnoty na standardní výstup

Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **`write`** jazyka IFJ21 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 IFJcode21. 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 jediný student. 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[20], 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 číslo týmu.

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

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 IS FIT 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 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 *číslo*, 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 překladače k nalezení.

Dokumentace nesmí:

- obsahovat kopii zadání či text, obrázky[21] 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 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

[21] Vyjma obyčejného loga fakulty na úvodní straně.

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 IFJcode21 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 *Souborech* předmětu v IS FIT je k dispozici 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, wiki stránkách 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 wiki stránkách 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 wiki stránkách 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 speciálního termínu „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 následně obdrží jeho vyhodnocení a 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 wiki stránky 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 IFJ21

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

- BOOLTHEN: Podpora typu **`boolean`**, booleovských hodnot **`true`** a **`false`**, booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**`not`**, **`and`**, **`or`** včetně zkratového vyhodnocení), jejichž priorita a asociativita odpovídá jazyku Teal. Pravdivostní hodnoty lze porovnávat jen operátory **`==`** a **`~=`**. Dále podporujte výpisy hodnot typu **`boolean`** a přiřazování výsledku booleovského výrazu do proměnné. Dále podporujte zjednodušený podmíněný příkaz **`if`** bez části **`else`** a rozšířený podmíněný příkaz s volitelným vícenásobným výskytem části **`elseif`** (+1,5 bodu).
- CYCLES: Překladač bude podporovat i cyklus **`for`** (numerickou variantu) a cyklus **`repeat`**-**`until`**. Dále bude podporovat ve všech typech iterace klíčové slovo **`break`** (+1,5 bodu).
- FUNEXP: Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce (+1,5 bodu).
- OPERATORS: Rozšíření zavádí podporu pro unárním operátor – (unární mínus) a binární aritmetické operátory % a ^. Sémantika, priorita a asociativita těchto operátorů odpovídá jazyku Teal. Dále je možné ve zdrojovém kódu IFJ21 používat záporné číselné literály (tj. jako term) (+0,5 bodu).

13 Opravy zadání

- 24. 9. 2021 – Oprava překlepů v příkladech (print→write); bílé znaky mezi příkazy nejsou vždy nezbytné.
- 25. 9. 2021 – Vypuštění klíčových slov read a write (identifikátory vestavěných funkcí nejsou klíčová slova); úprava sémantiky příkazu volání funkce při špatném počtu vracených hodnot; oprava překlepů a upřesnění formulací.
- 28. 9. 2021 – Doplnění požadavku na dokumentaci ohledně popisu rozdělení implementace na zdrojové soubory; oprava/doplnění sémantiky vestavěných funkcí (především těch pro práci s řetězci); oprava příkladu 1 a 3; doplnění rozšíření OPERATORS.