ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL

Zbyněk Křivka, Lukáš Zobal, Dominika Regéciová
email: {krivka, izobal, iregeciova}@fit.vut.cz
23. září 2020

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ20.
Informace: diskuzní fórum a wiki stránky předmětu IFJ v IS FIT.
Pokusné odevzdání: čtvrtek 26. listopadu 2020, 23:59 (nepovinné).
Datum odevzdání: středa 9. prosince 2020, 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 IFJ20 a přeloží jej do cílového jazyka IFJcode20 (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, neočekávané odřádkování).
- 3 - sémantická chyba v programu – nedefinovaná funkce/proměnná, pokus o redefinici funkce/proměnné, atp.
- 4 - sémantická chyba při odvozování datového typu nově definované proměnné.
- 5 - sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.
- 6 - sémantická chyba v programu – špatný počet/typ parametrů či návratových hodnot u volání funkce či návratu z funkce.
- 7 - ostatní sémantické chyby.
- 9 - sémantická chyba dělení nulovou konstantou.
- 99 - interní chyba překladače tj. neovlivněná vstupním programem (např. chyba alokace paměti, atd.).

Překladač bude načítat řídicí program v jazyce IFJ20 ze standardního vstupu a generovat výsledný mezikód v jazyce IFJcode20 (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 IFJcode20 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 IFJ20 je zjednodušenou podmnožinou jazyka Go[2], což je staticky typovaný[3] imperativní jazyk.

3.1 Obecné vlastnosti a datové typy

V programovacím jazyce IFJ20 **záleží** na velikosti písmen u identifikátorů i klíčových slov (tzv. *case-sensitive*). V lexémech (identifikátorech, literálech) neuvažujte vícebajtové znaky kódování Unicode (např. UTF-8).

- *Identifikátor* je definován jako neprázdná posloupnost písmen, číslic a znaku podtržítka ('_') začínající písmenem nebo podtržítkem.
- Jazyk IFJ20 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam, a proto se nesmějí vyskytovat jako identifikátory[4]:

  `else`, **`float64`**, **`for`**, **`func`**, **`if`**, **`int`**, **`package`**, **`return`**, **`string`**.

- *Celočíselný literál* (rozsah 64 bitů[5]) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě[6].
- *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)[7].
- *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 Go[8]. Neexistující escape sekvence vede na lexikální chybu. Znak v řetězci může být zadán také pomocí obecné hexadecimální escape sekvence '**`\x`***hh*', kde *hh* je právě dvoumístné hexadecimální číslo od **`00`** do **`FF`** (písmena `A-F` mohou být malá nebo velká).
  Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

[2] `https://golang.org/doc/`; na serveru Merlin je pro studenty k dispozici překladač `go` verze 1.15.

[3] Jednotlivé proměnné mají datový typ určen deklarací nebo staticky odvozen na základě zdrojového kódu.

[4] Nekoliznost je třeba zajistit i vůči identifikátorům vestavěných funkcí. V rozšířeních potom mohou být použita i další klíčová slova, která ale budeme testovat pouze v případě implementace patřičného rozšíření.

[5] Pro implementaci v jazyce C lze využít například typ **`int64_t`** ze standardu C99.

[6] Přebytečná počáteční číslice `0` není povolena, protože značí zápis čísla v jiné číselné soustavě.

[7] Přebytečné počáteční číslice `0` v celočíselné části jsou chybou, v exponentu jsou ignorovány.

[8] `https://golang.org/ref/spec#Rune_literals`

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

reprezentuje řetězec
Ahoj
"Sve'te \".

- *Datové typy* nejen pro ukládání literálů jsou označeny **int**, **float64** a **string**. Proměnné číselných datových typů mohou obsahovat i záporné hodnoty.
- *Term* je libovolný literál (celočíselný, desetinný či řetězcový) nebo identifikátor proměnné.
- Jazyk IFJ20 podporuje *řádkové* i *blokové* komentáře stejně jako jazyk Go. Řádkový komentář začíná dvojicí lomítek ('**//**', ASCII hodnoty 47) a za komentář je považováno vše, co následuje do konce řádku kromě samotného odřádkování. Blokový komentář začíná dvojicí symbolů '**/***' a je ukončen první následující dvojicí symbolů '***/**', takže hierarchické vnoření blokových komentářů není podporováno.

4 Struktura jazyka

IFJ20 je strukturovaný programovací jazyk podporující definice proměnných a uživatelských funkcí včetně jejich rekurzivního volání. Vstupním bodem prováděného programu je povinná *hlavní funkce* `main`.

4.1 Základní struktura jazyka

Program se skládá z prologu následovaného sekvencí definic uživatelských funkcí včetně definice hlavní funkce `main`. Chybějící funkce `main` způsobí chybu 3. Funkce `main` nemá definován žádný parametr ani návratový typ, jinak chyba 6.

Prolog[9] se skládá z jednoho řádku:

```
package main
```

Na každém řádku těla definice funkce se může nacházet nejvýše jeden příkaz jazyka IFJ20. Mezi příkazy se může vyskytovat libovolný počet bílých znaků (mezera, tabulátor, komentář a odřádkování). Na začátku a konci zdrojového textu se též smí vyskytovat libovolný počet bílých znaků. Mezi lexémy, které nejsou na rozmezí dvou příkazů, jsou povoleny neodřádkovací bílé znaky (tj. mezery, tabulátory a blokové komentáře). Další možnosti vkládání odřádkování (potažmo řádkových komentářů) jsou specifikovány u příkazů/výrazů v sekcích 4.3 a 5.1. Struktura definice uživatelských funkcí a také popis jednotlivých příkazů je v následujících sekcích.

4.1.1 Proměnné

Proměnné jazyka IFJ20 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

[9] Prolog mohou prokládat komentáře a prázdné řádky a slouží především kvůli kompatibilitě s programy jazyka Go.

příkazů zapsána ve složených závorkách v rámci podmíněného příkazu, příkazu cyklu a těla funkce (tj. na rozdíl od jazyka Go nelze vytvářet blok samostatnými složenými závorkami).

Definice proměnné se provádí pomocí příkazu definice proměnné s povinnou inicializací a odvozením datového typu pomocí operátoru '`:=`'. Detailní syntaxe bude popsána v sekci 4.3.

Nelze-li při definici proměnné z povinného inicializačního výrazu staticky (při překladu) odvodit typ, jde o chybu 4. Výraz, jehož typ lze odvodit staticky, může obsahovat literály, již definované proměnné, operátory i závorky. Nelze definovat proměnnou stejného jména, jako má jiná proměnná na stejné úrovni (chyba 3). Proměnná může být definována stejného jména jako již definovaná funkce, pak v daném rozsahu platnosti tuto funkci překryje, a tak v tomto rozsahu není možné kolizní funkci volat. 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 Definice uživatelských funkcí

Definice funkce se skládá z hlavičky a těla funkce. Každá uživatelská funkce s daným identifikátorem je definována nejvýše jednou[10], jinak dochází k chybě 3. Definice funkce nemusí vždy lexikálně předcházet kódu pro volání této funkce[11]. Uvažujte například vzájemné rekurzivní volání funkcí (tj. funkce f volá funkci g, která opět může volat funkci f).

Definice uživatelské funkce je následujícího tvaru:

- *Definice funkce* je víceřádková konstrukce (hlavička a tělo) ve tvaru:
  `func` *id* **`(`** *seznam_parametrů* **`)`** **`(`** *seznam_návratových_typů* **`)`** **`{`**
  složený_příkaz
  `}`
  - Hlavička definice funkce (od klíčového slova **`func`** až '`{`' včetně) nesmí obsahovat odřádkování[12].
  - 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 až potom[13] je jeho datový typ: *identifikátor_parametru typ*
    Parametry jsou vždy předávány hodnotou.
  - Seznam návratových typů je posloupnost datových typů, které funkce vrací. Oddělovačem v seznamu návratových typů je opět čárka. Příklad:

[10] Tzv. přetěžování funkcí (angl. *overloading*) není v IFJ20 podporováno.

[11] Typické řešení vyžaduje dvouprůchodovou syntaktickou analýzu. V prvním průchodu si poznačíte signatury definovaných funkcí (signatura zahrnuje identifikátor a typy parametrů a návratových hodnot), takže při druhém průchodu budete vědět, zda bude nějaká funkce ve zdrojovém kódu definována později a s jakou signaturou.

[12] V Go 1.15 ve skutečnosti smí být odřádkování v seznamu parametrů a v seznamu návratových typů, a to za oddělující čárkou, což je součástí rozšíření FUNEXP.

[13] Pojednání o zápisu typů v Go: `https://blog.golang.org/declaration-syntax`

```
func swap (x string, y string) (string, string) {
    return y, x
}

func main() {
}
```

Seznam návratových typů může být i prázdný a v takovém případě může být celý (včetně kulatých závorek) vynechán (viz např. funkce `main`).

- Tělo funkce je tvořeno sekvencí příkazů (viz sekce 4.3) zapsaných ve složených závorkách. Parametry funkce jsou chápány jako předdefinované lokální proměnné.
- V Go je konvence, že funkce přístupné mimo daný balík musí začínat velkým písmenem. Jelikož IFJ20 podporuje pouze jediný balík *main*, tak budeme vestavěné funkce začínat malými písmeny. Uživatelské funkce mohou začínat velkými i malými písmeny.

Má-li funkce neprázdný seznam návratových typů, tak vrací hodnotu/hodnoty dané vyhodnocením výrazu/výrazů v příkazu **`return`**. V případě chybějící návratové hodnoty takovéto funkce kvůli neprovedení žádného příkazu **`return`** dojde k chybě 6. Stejně tak dojde k chybě 6, pokud **`return`** vrací hodnoty, které neodpovídají seznamu návratových typů. 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 a vždy je ohraničena složenými závorkami, které jsou v popisu syntaxe explicitně uvedeny.
Dílčím příkazem se rozumí:

- *Příkaz definice proměnné*:
  id **`:=`** *výraz*
  Sémantika příkazu je následující: Příkaz nově definuje proměnnou, která v daném bloku ještě nesmí být definována (jinak chyba 3). Typ nově definované proměnné bude staticky (při překladu) odvozen od typu výrazu *výraz*. Všimněte si, že daná proměnná mohla být definována v nadřazeném bloku a tímto příkazem bude překryta (tj. ve zbytku současného bloku nebude přístupná/viditelná, ačkoli je stále platná).
- *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 zleva doprava), zapamatování si těchto výsledků a následně přiřazení těchto výsledků jednotlivým proměnným. Všechny proměnné nalevo od **`=`** musí být dříve definované. Speciální proměnnou je _,

kterou nelze definovat a lze do ní přiřadit libovolná hodnota, kterou ale nelze přečíst. Prakticky je taková hodnota zahozena, aniž bychom k tomu účelu museli definovat pomocnou proměnnou.

- *Podmíněný příkaz*:
  if *výraz* **{**
  sekvence_příkazů$_1$
  } else {
  sekvence_příkazů$_2$
  }
  Mezi *výraz* a '**{**' a mezi '**}**' a **else** se nesmí vyskytovat odřádkování. Možnosti odřádkování ve výrazu *výraz* jsou popsány v sekci 5. 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 nebo nepravda – v základním zadání pouze jako výsledek aplikace relačních operátorů dle sekce 5.1), nastává chyba 5.
- *Příkaz cyklu*:
  for *definice* **;** *výraz* **;** *příkaz_přiřazení* **{**
  sekvence_příkazů
  }
  Příkaz cyklu se skládá z hlavičky a těla tvořeného *sekvencí_příkazů*.
  Sémantika příkazu cyklu je následující: Před provedením první iterace může být definována lokální proměnná (tzv. iterační) včetně inicializace výrazem v rámci *definice* (při vynechání se uvede pouze oddělující středník). Před provedením těla cyklu je vždy vyhodnocena podmínka *výraz* a v případě pravdivosti je provedeno tělo cyklu. Na konci prováděného těla cyklu je vykonáno přiřazení pomocí *příkaz_přiřazení*, kdy se typicky modifikuje iterační proměnná nebo jinak ovlivňuje pravdivost podmínky pro vykonání další iterace. Pak následuje opět vyhodnocení a kontrola pravdivosti podmínky pro případné opětovné provedení těla cyklu. Pravidla pro určení pravdivosti výrazu jsou stejná jako u výrazu podmíněného příkazu. Nově definovaná iterační proměnná je platná v hlavičce i následném těle příkazu cyklu, nicméně její výskyt například v podmínce *výraz* není povinný. Pozor, že proměnnou stejného jména jako iterační proměnná lze v *sekvenci_příkazů* znovu definovat, a tím překrýt iterační proměnnou. Stejně jako část *definice*, není povinná ani část *příkaz_přiřazení*, ale oddělující středníky musí zůstat zachovány. Část *výraz* je povinná. Proměnné definované až v těle cyklu nejsou platné v hlavičce cyklu a proměnné definované v hlavičce cyklu nejsou platné za tělem cyklu.
- *Volání vestavěné či uživatelem definované funkce*:
  název_funkce **(***seznam_vstupních_parametrů***)**
  nebo
  id_1, id_2, …, id_n **=** *název_funkce* **(***seznam_vstupních_parametrů***)**
  kde $n \geq 1$. První varianta slouží pouze pro volání funkcí bez návratové hodnoty. U druhé varianty slouží čárka ('**,**') syntakticky jako oddělovač při $n \geq 2$, takže za posledním identifikátorem se čárka nepíše.
  Seznam_vstupních_parametrů je seznam termů (viz sekce 3.1) oddělených čárka-

mi[14]. 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 6. 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. Proměnná id_i může být i speciální proměnná _, která vyjadřuje, že odpovídající návratová hodnota bude zahozena. Máme-li funkci s neprázdným seznamem návratových typů, jejíž tělo neobsahuje žádný příkaz[15] **return**, jedná se o chybu 6. Všechny proměnné nalevo od **=** již musí být definovány.

- *Příkaz návratu z funkce*:
  return *seznam_výrazů*
  Příkaz musí být použit v těle každé funkce, která nemá prázdný seznam návratových typů. U funkcí s prázdným seznamem návratových typů (např. `main`) může být příkaz návratu z funkce zcela vynechán nebo použita varianta bez seznamu výrazů. Jeho sémantika je následující: Dojde k vyhodnocení (zleva doprava) jednotlivých čárkou oddělených výrazů v *seznam_výrazů* (tj. získání n-tice návratových hodnot, kde $n \geq 0$), okamžitému ukončení provádění těla funkce a návratu do místa volání, kam funkce vrátí vypočtenou návratovou hodnotu/hodnoty. Je-li počet výrazů či typy výsledných hodnot nekompatibilní s návratovými typy dané funkce, dojde k chybě 6.

5 Výrazy

Výrazy jsou tvořeny termy, závorkami a binárními aritmetickými, řetězcovým a relačními operátory. Odřádkování lze vložit jen za operátor či otevírací závorku, ale nebudeme vkládání odřádkování do výrazů testovat.

IFJ20 je silně typovaný, takže nejsou prováděny žádné[16] implicitní konverze operandů či výsledků výrazů či funkcí.

Pro chybné kombinace datových typů vracejte chybu 5.

5.1 Aritmetické, řetězcové a relační operátory

Standardní binární operátory **+**, **-**, ***** značí sčítání, odčítání[17] a násobení. Jsou-li oba operandy typu **int**, je i výsledek typu **int**. Jsou-li oba operandy typu **float64**, výsledek je též typu **float64**. Operátor **+** navíc provádí se dvěma operandy typu **string** jejich konkatenaci.

Operátor **/** značí dělení dvou číselných operandů. Jsou-li oba operandy celočíselné, jedná se o celočíselné dělení. Jsou-li oba operandy desetinné, značí **/** dělení v plovoucí

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

[15] Jazyk Go má kontroly přísnější a sleduje i tok řízení uvnitř funkce, což ale není vaším úkolem.

[16] Na rozdíl od jazyka Go, kde je podporována implicitní konverze číselných literálů (nikoli proměnných či složitějších výrazů) z typu **int** na **float64**.

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

řádové čárce. Při dělení nulou operátorem `/` dochází k běhové[18] chybě 9. Pro provedení explicitního přetypování mezi **`int`** a **`float64`** lze použít vestavěné funkce **`int2float`** a **`float2int`**.

Pro relační operátory `<`, `>`, `<=`, `>=`, `==`, `!=` platí, že výsledkem porovnání je pravdivostní hodnota a že mají stejnou sémantiku jako v jazyce Go. Tyto operátory pracují s operandy stejného typu, a to **`int`**, **`float64`** nebo **`string`**. U řetězců se porovnání provádí lexikograficky. Bez rozšíření BOOLTHEN není s výsledkem porovnání možné dále pracovat a lze jej využít pouze u příkazů **`if`** a **`for`**.

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	`<` `<=` `>` `>=` `==` `!=`	levá[19]

6 Vestavěné funkce

Překladač bude poskytovat některé základní vestavěné funkce, které bude možné využít v programech jazyka IFJ20. Pro generování kódu vestavěných funkcí lze výhodně využít specializovaných instrukcí jazyka IFJcode20.

Při použití špatného typu termu v parametrech následujících vestavěných funkcích dochází k chybě 6.

Vestavěné funkce pro načítání literálů a výpis termů:

- **`func inputs() (string,int)`**
  `func inputi() (int,int)`
  `func inputf() (float64,int)`
  Vestavěné funkce ze standardního vstupu načtou jeden řádek ukončený odřádkováním. Funkce **`inputs`** tento řetězec vrátí bez symbolu konce řádku (načítaný řetězec nepodporuje escape sekvence). V případě **`inputi`** a **`inputf`** jsou okolní bílé znaky ignorovány. Jakýkoli jiný nevhodný znak před samotným číslem je známkou špatného formátu a vede na chybu (hodnota 1 v druhé návratové hodnotě). Funkce **`inputi`** načítá a vrací celé číslo, **`inputf`** 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 nebo jejího špatného formátu bude druhá návratová hodnota **`1`**.

[18] Při rozpoznání dělení nulou jakožto konstantou dojde k chybě 9 již při překladu.

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

- *Příkaz pro výpis hodnot*:
  func print ($term_1$ **,** $term_2$ **, …,** $term_n$ **)**
  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'`[20], hodnota termu typu float64 pak pomocí `'%a'`[21]. Funkce **print** nemá návratovou hodnotu.

Vestavěné funkce pro konverzi číselných typů:

- **func int2float(**i **int) (float64)** – Vrací hodnotu celočíselného parametru *i* převedenou na desetinnou hodnotu.
- **func float2int(**f **float64) (int)** – Vrací hodnotu desetinného parametru *f* převedenou na celočíselnou hodnotu oříznutím desetinné části.

Vestavěné funkce pro práci s řetězci: Případná druhá návratová hodnota je věnována *příznaku chyby*, kdy 0 znamená bezchybné provedení příkazu.

- **func len(***s* **string) (int)** – Vrátí délku (počet znaků) řetězce zadaného jediným parametrem *s*. Např. **len("x\nz")** vrací 3.
- **func substr(**s **string,** i **int,** n **int) (string, int)** – Vrátí podřetězec zadaného řetězce *s*. Druhým parametrem *i* je dán začátek požadovaného podřetězce (počítáno od nuly) a třetí parametr *n* určuje délku podřetězce. Je-li index *i* mimo meze 0 až **len(**s**)** nebo $n < 0$, vrací funkce v příznaku chyby hodnotu 1. Je-li $n >$ **len(**s**)** $- i$, jsou jako řetězec vráceny od *i*-tého znaku všechny zbývající znaky řetězce *s*.
- **func ord(**s **string,** i **int) (int, int)** – Vrátí ordinální hodnotu (ASCII) znaku na pozici *i* v řetězci *s*. Je-li index *i* mimo meze řetězce (0 až **len(**s**)** - 1), vrací funkce v příznaku chyby hodnotu 1.
- **func chr(**i int**) (string, int)** – Vrátí jednoznakový řetězec se znakem, jehož ASCII kód je zadán parametrem *i*. Případ, kdy je *i* mimo interval $[0; 255]$, vede na hodnotu 1 v příznaku chyby.

7 Implementace tabulky symbolů

Tabulka symbolů bude implementována pomocí abstraktní datové struktury, která je ve variantě zadání pro daný tým označena velkými římskými číslicemi, 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.

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

8 Příklady

Tato kapitola uvádí tři jednoduché příklady řídicích programů v jazyce IFJ20.

8.1 Výpočet faktoriálu (iterativně)

```
// Program 1: Vypocet faktorialu (iterativne)
package main

func main() {
    print("Zadejte cislo pro vypocet faktorialu: ")
    a := 0
    a, _ = inputi()
    if a < 0 {
        print("Faktorial nejde spocitat!\n")
    } else {
        vysl := 1
        for ; a > 0; a = a - 1 {
            vysl = vysl * a
        }
        print("Vysledek je ", vysl, "\n")
    }
}
```

8.2 Výpočet faktoriálu (rekurzivně)

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

func factorial(n int) (int) {
    dec_n := n - 1
    if n < 2 {
        return 1
    } else {
        tmp := 0
        tmp = factorial(dec_n)
        return n * tmp
    }
}

func main() {
    print("Zadejte cislo pro vypocet faktorialu: ")
    a := 0
    err := 0
    a, err = inputi()
    if err == 0 {
        if a < 0 {
            print("Faktorial nejde spocitat!", "\n")
        } else {
            vysl := 0
            vysl = factorial(a)
            print("Vysledek je ", vysl, "\n")
        }
    } else {
```

```
        print("Chyba pri nacitani celeho cisla!\n")
    }
}
```

8.3 Práce s řetězci a vestavěnými funkcemi

```
// Program 3: Prace s retezci a vestavenymi funkcemi
package main

func main() {
    s1 := "Toto je nejaky text"
    s2 := s1 + ", ktery jeste trochu obohatime"
    print(s1, "\n", s2)
    s1len := 0
    s1len = len(s1)
    s1len = s1len - 4
    s1, _ = substr(s2, s1len, 4)
    s1len = s1len + 1
    print("4 znaky od", s1len, ". znaku v \"", s2, "\":", s1, "\n")
    print("Zadejte serazenou posloupnost vsech malych pismen a-h, ")
    print("pricemz se pismena nesmeji v posloupnosti opakovat: ")
    err := 0
    s1, err = inputs()
    if err != 1 {
        for ;s1 != "abcdefgh"; {
            print("\n", "Spatne zadana posloupnost, zkuste znovu: ")
            s1, _ = inputs()
        }
    } else {
    }
}
```

9 Doporučení k testování

Programovací jazyk IFJ20 je schválně navržen tak, aby byl téměř kompatibilní s podmnožinou jazyka Go[22]. Pokud si student není jistý, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ20, může si to ověřit následovně. Z IS FIT si stáhne ze *Souborů* k předmětu IFJ ze složky *Projekt* soubor `ifj20.go` obsahující kód, který doplňuje kompatibilitu IFJ20 s překladačem `go` jazyka Go 1.15 na serveru `merlin`. Soubor `ifj20.go` obsahuje definice vestavěných funkcí, které jsou součástí jazyka IFJ20, ale chybí v potřebné formě v jazyce Go.

Váš program v jazyce IFJ20 uložený například v souboru `testPrg.go` (přípona `go` je vyžadována překladačem jazyka Go) pak lze provést na serveru `merlin` například pomocí příkazu:

```
go run testPrg.go ifj20.go < test.in > test.out
```

Tím lze jednoduše zkontrolovat, co by měl provést zadaný zdrojový kód přeložený na cílový kód. Je ale potřeba si uvědomit, že jazyk Go je nadmnožinou jazyka IFJ20, a

[22] Online dokumentace ke Go: `https://golang.org/doc/`

tudíž může zpracovat i konstrukce, které nejsou v IFJ20 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 IFJcode20

Cílový jazyk IFJcode20 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 IFJcode20 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode20 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode20
```

10.1 Hodnotící interpret `ic20int`

Pro hodnocení a testování mezikódu v IFJcode20 je k dispozici interpret pro příkazovou řádku (`ic20int`):

```
ic20int 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 IFJcode20.
- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode20.
- 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. IFJcode20 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 IFJcode20 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 IFJ20.

Zápis každé konstanty v IFJcode20 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*⟩$\in$ {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 **`inputs`**, **`inputi`** a **`inputf`** jazyka IFJ20 a **`inputb`** v rámci rozšíření BOOLTHEN.

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

Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **`print`** jazyka IFJ20 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 IFJcode20. 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[23], 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.

11.2 Dělení bodů

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

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

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.

Dokumentace nesmí:

- obsahovat kopii zadání či text, obrázky[24] 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; překážky, 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 archivu 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 IFJcode20 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

[24] Vyjma obyčejného loga fakulty na úvodní straně.

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

V průběhu řešení (do stanoveného termínu) bude postupně (případně i na váš popud) aktualizován ceník rozšíření a identifikátory rozšíření projektu (viz 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 IFJ20

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

- BOOLTHEN: Podpora typu **`bool`**, booleovských hodnot **`true`** a **`false`**, booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**`!`**, **`&&`**, **`||`**), jejichž priorita a asociativita odpovídá jazyku Go. Pravdivostní hodnoty lze porovnávat jen operátory **`==`** a **`!=`**. Dále podporujte načítání vestavěnou funkcí `inputb` a výpisy hodnot typu **`bool`** 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 **`else if`** (+1,5 bodu).
- BASE: Celočíselné konstanty je možné zadávat i ve dvojkové (číslo začíná znaky '**`0b`**'), osmičkové (číslo začíná nulou a písmenem o, '**`0o`**') a v šestnáctkové (číslo začíná znaky '**`0x`**') soustavě (+0,5 bodu). Místo malých písmen '**`b`**', '**`o`**' a '**`x`**', lze použít i velká písmena '**`B`**', '**`O`**' a '**`X`**'. Navíc lze mezi číslicemi použít znak podtržítko ('_') pro přehlednější zápis velkých čísel dle pravidel v jazyce Go.
- FUNEXP: Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce. Vyhodnocování hodnot parametrů při volání funkce uvažujte zleva doprava. Na rozdíl od základního zadání je třeba podporovat odřádkování uvnitř závorek pro parametry funkce a pro návratové typy funkce (za oddělujícími čárkami). Dále je třeba podporovat odřádkování ve výrazech dle jazyka Go (+1,5 bodu).
- MULTIVAL: Proměnné lze definovat pomocí id_1, id_2, …, id_n **`:=`** $výraz_1$, $výraz_2$, …, $výraz_n$, kde $n \geq 1$. Příkaz má stejné vyhodnocování jako příkaz přiřazení, přičemž alespoň jedna proměnná musí být nově definována (speciální proměnná _ se nepočítá). Proměnným již definovaným v daném bloku se pouze přiřadí nová hodnota. V případě, že jsou všechny proměnné již definované, dojde k chybě 3. Součástí rozšíření jsou i pojmenované návratové hodnoty:

```
func id ( seznam_parametrů ) ( seznam_pojmenovaných_návratových_typů ) {
    ⋮
  return
}
```

kde *seznam_pojmenovaných_návratových_typů* má stejnou podobu jako *seznam_parametrů*, tedy seznam definic *identifikátor_návratové_hodnoty typ* oddělených čárkou. Identifikátory jsou chápány jako lokální proměnné, které mají na začátku implicitní hodnoty (0 pro **int**, 0.0 pro **float64**, **""** pro **string**). Tělo funkce poté obsahuje povinnou část **return**, již bez seznamu výrazů (+1,5 bodu).

- UNARY: Rozšíření zavádí podporu pro unární operátor − (unární mínus) a + (unární plus) a přiřazovací příkazy **+=**, **-=**, ***=**, **/=**. Priorita a asociativita unárního mínus/plus a také syntaxe a sémantika nových přiřazovácích příkazů odpovídá jazyku Go (+0,5 bodu).

13 Opravy zadání

- 24. 9. 2020 – Oprava řady překlepů/pravopisných chyb a oprava příkladů.
- 25. 9. 2020 – **package** přidáno do seznamu klíčových slov. Doplnění rozšíření BOOLTHEN, přidání rozšíření BASE a UNARY. Oprava drobných chyb.
- 28. 9. 2020 – Doplnění rozšíření FUNEXP, popisu signatury funkce `main` a volání funkce bez návratové hodnoty, povinná přípona `.go`, opravy překlepů.