ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL

Zbyněk Křivka, Lukáš Zobal, Ľubica Genčúrová
email: {krivka, izobal, igencurova}@fit.vut.cz
23. září 2019

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ19.
Informace: diskuzní fórum a wiki stránky předmětu IFJ v IS FIT.
Pokusné odevzdání: pátek 29. listopadu 2019, 23:59 (nepovinné).
Datum odevzdání: středa 11. prosince 2019, 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 IFJ19 a přeloží jej do cílového jazyka IFJcode19 (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, špatný počet mezer odsazení).
- 2 - chyba v programu v rámci syntaktické analýzy (chybná syntaxe programu, neočekávaná změna úrovně odsazení).
- 3 - sémantická chyba v programu – nedefinovaná funkce/proměnná, pokus o redefinici funkce/proměnné, atp.
- 4 - sémantická/běhová chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech.
- 5 - sémantická chyba v programu – špatný počet parametrů u volání funkce.
- 6 - ostatní sémantické chyby.
- 9 - běhová chyba dělení nulou.
- 99 - interní chyba překladače tj. neovlivněná vstupním programem (např. chyba alokace paměti, atd.).

Překladač bude načítat řídicí program v jazyce IFJ19 ze standardního vstupu a generovat výsledný mezikód v jazyce IFJcode19 (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 IFJcode19 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 IFJ19 je zjednodušenou podmnožinou jazyka Python 3[2], což je dynamicky typovaný[3] imperativní (objektově-orientovaný) jazyk s funkcionálními prvky.

3.1 Obecné vlastnosti a datové typy

V programovacím jazyce IFJ19 **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.
- Jazyk IFJ19 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam, a proto se nesmějí vyskytovat jako identifikátory[4]:

  `def`, **`else`**, **`if`**, **`None`**, **`pass`**, **`return`**, **`while`**.

- *Celočíselný literál* (rozsah C-int) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě[5].
- *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)[6].
- *Řetězcový literál* je oboustranně ohraničen jednoduchým apostrofem (`'`, ASCII hodnota 39). Tvoří jej libovolný počet znaků zapsaných na jediném řádku programu. Možný je i prázdný řetězec (`''`). Znaky s ASCII hodnotou větší než 31 (mimo `'` a `\`) lze zapisovat přímo. Některé další znaky lze zapisovat pomocí escape sekvence: '**`\"`**', '**`\'`**', '**`\n`**', '**`\t`**', '**`\\`**'. Jejich význam se shoduje s odpovídajícími znakovými konstantami jazyka Python 3[7]. 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`** (pro hexadecimální číslice `A-F` podporujte i malá písmena). Při použití nedefinované escape sekvence (např. '**`\Z`**') je do výsledného řetězce vložena dvojice znaků[8] (lomítko a `Z`), jako by se o escape sekvenci nejednalo[9].
  Délka řetězce není omezena (resp. jen dostupnou pamětí). Například řetězcový literál

[2] `https://docs.python.org/3/reference/`; na serveru Merlin je pro studenty k dispozici interpret a interaktivní konzole `python3` verze 3.6.9.

[3] Jednotlivé proměnné mají datový typ určen hodnotou/objektem, který obsahují.

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

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

[7] `https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals`

[8] Výjimkou je hexadecimální escape sekvence, kde nedodržení formátu vede na chybu 1.

[9] Escape sekvence definované v Python 3, ale ne v IFJ19, nebudou testovány.

```
'Ahoj\n"Sve\'te \\\x27'
```

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

- *Dokumentační řetězec* je literál podobně jako základní řetězcový, ale je oboustranně ohraničen třemi uvozovkami '**"""**' (uvozovka má ASCII hodnotu 34) a může obsahovat konce řádků (případné řádkové komentáře jsou součástí výsledného dokumentačního řetězce), které však neovlivňují syntaktickou strukturu příkazů (viz sekce 4). Kromě hrozícího vzniku nejednoznačnosti s ukončovací trojicí uvozovek není třeba využívat escape sekvence a většinu znaků lze zapisovat přímo.
- Literál **None** zastupuje speciální hodnotu (tzv. neznámá hodnota), která z hlediska IFJ19 nemá typ[10].
- *Datové typy* pro jednotlivé literály jsou označeny int, float a str. Typy se využívají pouze interně a mají význam u sémantických kontrol.
- *Term* je libovolný literál (celočíselný, desetinný či řetězcový), hodnota **None** nebo identifikátor proměnné.
- Jazyk IFJ19 podporuje *řádkové* komentáře stejně jako Python. Řádkový komentář začíná znakem mřížka (' # ', ASCII hodnota 35) a za komentář je považováno vše, co následuje až do konce řádku. Pro zakomentování větší části programu je možné stejně jako v Pythonu využít dokumentační řetězec.

4 Struktura jazyka

IFJ19 je strukturovaný programovací jazyk podporující definice proměnných a uživatelských funkcí, základní řídicí příkazy, příkaz přiřazení a volání funkce včetně rekurzivního. Vstupním bodem řídicího programu je neoznačená nesouvislá sekvence příkazů mezi definicemi uživatelských funkcí, tzv. *hlavní tělo* programu.

4.1 Základní struktura jazyka

Program se skládá ze sekvence definic uživatelských funkcí a příkazů. Příkazy mimo definice funkcí tvoří hlavní tělo programu. V těle definice funkce i v hlavním těle programu se může nacházet libovolný počet příkazů jazyka IFJ19.

Jednotlivé konstrukce jazyka IFJ19 jsou (až na výjimky) jednořádkové[11] a jako takové jsou odděleny znakem konce řádku (ASCII hodnota 10, dále jen **EOL**). U definic víceřádkových konstrukcí jsou dodatečné znaky **EOL** explicitně uvedeny[12].

[10] V Python 3 je hodnota None instance třídy NoneType.

[11] Mezi závorkami (tedy ve výrazech a mezi parametry funkcí), kdy se v Python 3 nesleduje odsazení nového řádku, není třeba v IFJ19 podporovat vkládání **EOL**.

[12] Vně zmíněných konstrukcí je znak **EOL** brán jako tzv. *bílý znak*. Je tedy například možné pro zpřehlednění kódu vložit mezi dva příkazy prázdný řádek.

4.2 Odsazení a bílé znaky

Počet mezer[13] na začátku řádku udává *odsazení* příkazu na daném řádku, což umožňuje podle *úrovně odsazení* seskupovat příkazy do sekvence příkazů, takže není třeba mít v syntaxi IFJ19 složené závorky jako například v jazyce C.

Pro zjednodušení analýzy odsazení řádků zdrojového textu v rámci lexikální analýzy je výhodné s využitím pomocného zásobníku (doplňuje deterministický konečný automat v lexikálním analyzátoru) generovat pouze speciální tokeny pro zvýšení nebo snížení úrovně odsazení (*INDENT* a *DEDENT*). Před začátkem analýzy se na tento zásobník umístí pomocná 0, která nebude nikdy vyjmuta. Při analýze každého řádku je zjištěno jeho odsazení (tj. počet mezer před prvním nebílým znakem, který není začátkem řádkového komentáře[14]) a porovnáno s hodnotou na vrcholu zásobníku. (a) Je-li vyšší, vložíme novou hodnotu na zásobník a generujeme token *INDENT*, (b) je-li stejné, negenerujeme žádný speciální token, (c) při nižší hodnotě vyjímáme ze zásobníku tak dlouho, dokud nenarazíme na tuto hodnotu (nenajde-li se, jedná se o špatný počet mezer odsazení a chybu 1) a při každém vyjmutí generujeme token *DEDENT*. Na konci zdrojového souboru dogenerujeme tokeny *DEDENT* za každou zbylou nenulovou hodnotu na pomocném zásobníku.[15]

V případě začátku sekvence příkazů typicky dává lexikální analýza speciální token *INDENT* a za koncem dané sekvence dává odpovídající token *DEDENT*, což nepřímo odpovídá například tokenům pro otevírací a zavírací složenou závorku v jazyce C. V následném popisu syntaxe IFJ19 budou změny úrovně odsazení naznačeny pouze intuitivně. V případě vygenerování speciálního tokenu tam, kde to syntaktická analýza neočekává, dochází k neočekávané změně úrovně odsazení a chybě 2.

Bílé znaky bez komentářů (tj. mezery nebo tabulátory) se mohou vyskytovat v libovolném množství mezi všemi lexémy (kromě speciálních tokenů *INDENT* a *DEDENT*), i na začátku a na konci zdrojového textu. Jednořádkové komentáře mohou být vloženy na samostatný řádek, za jednořádkové příkazy (i v rámci sekvencí příkazů) a u víceřádkových příkazů či definicí funkcí na koncích jednotlivých řádků. Dokumentační řetězec lze ve funkci víceřádkového komentáře vložit pouze na samostatný řádek a za koncem dokumentačního řetězce mohou být pouze bílé znaky a odřádkování.

4.3 Hlavní tělo programu

Hlavní tělo programu je neoznačená sekvence příkazů IFJ19, která se prolíná s definicemi funkcí (na nejvyšší úrovni příkazů, definice funkce tudíž nemůže narušit integritu příkazu, a to ani složeného). Hlavní tělo programu může být tvořeno i prázdnou sekvencí příkazů, kdy je pouze vrácena návratová hodnota programu (možné hodnoty viz kapitola 2). Celá sekvence je ukončena koncem zdrojového souboru. Struktura jednotlivých příkazů je popsána v následujících kapitolách.

[13] Pro jednoduchost nebudeme uvažovat odsazení pomocí tabulátorů, jež je v Pythonu podporováno.

[14] Řádky obsahující pouze bílé znaky a případný řádkový komentář negenerují žádný ze speciálních tokenů.

[15] Detailnější popis i s příklady viz `https://docs.python.org/3/reference/lexical_analysis.html#indentation`.

4.3.1 Definice proměnných

Proměnné jazyka IFJ19 jsou lokální (definované v těle funkce) a globální (definované v hlavním těle programu). Lokální proměnné a parametry funkcí mají rozsah v dané funkci, kde byly definovány (od místa jejich definice po konec funkce). Globální proměnná je pak platná od místa její definice po zbytek běhu programu. Jazyk IFJ19 neobsahuje specifický příkaz pro definici proměnné, ale definice proměnné proběhne v rámci vykonání prvního přiřazení hodnoty do proměnné (viz popis příkazů níže). V případě, že je definiční příkaz přiřazení za běhu skutečně proveden, dojde k inicializaci proměnné na hodnotu přiřazovaného výrazu.

Každá v programu použitá proměnná musí být definována, jinak se jedná o sémantickou chybu 3. Nelze definovat proměnnou stejného jména, jako má některá již definovaná funkce, a naopak nelze definovat funkce stejného jména jako některá již definovaná proměnná v hlavním těle programu.

Viditelnost globální proměnné může být v uživatelské funkci překryta stejnojmennou lokální proměnnou nebo formálním parametrem funkce. V takové funkci potom nelze v IFJ19 ke globální proměnné přistoupit (ani před samotnou definicí lokální proměnné, jinak chyba 3).

4.4 Definice uživatelských funkcí

Každá funkce musí být definována před jejím použitím tzv. *voláním funkce* (příkaz volání je definován níže). Při volání funkce z hlavního těla programu musí být funkce nejprve definovaná. Definice funkce však nemusí být lexikálně umístěna před jejím příkazem volání, pokud je volána z jiné funkce, která je má příkaz volání umístěn až za definicemi obou funkcí. Jsou tedy umožněny vzájemné rekurzivní volání dvou či více funkcí i bez deklarací funkcí. Definice funkce je následujícího tvaru:

- *Definice funkce* je víceřádková konstrukce (hlavička a tělo) ve tvaru:
  `def` *id* **`(`** *seznam_parametrů* **`) : EOL`**
  sekvence_příkazů
  - Definice jednotlivých formálních parametrů jsou odděleny čárkou (’**`,`**’), za poslední z nich se čárka neuvádí. Seznam může být i prázdný. Parametry jsou vždy předávány hodnotou.
  - Tělo funkce je o úroveň odsazená, neprázdná sekvence dílčích příkazů (viz sekce 4.5). V těle funkce jsou její parametry chápány jako předdefinované lokální proměnné. Výsledek funkce je dán provedeným příkazem návratu z funkce (viz sekce 4.5).

Z hlediska sémantiky je potřeba kontrolovat, zda souhlasí počet parametrů v hlavičce definice funkce a v příkazech volání této funkce. Redefinice funkcí a přetěžování funkcí není povoleno.

4.5 Syntaxe a sémantika příkazů

Dílčím příkazem se rozumí:

- *Příkaz přiřazení*:
  id **=** *výraz*
  Sémantika příkazu je následující: Příkaz provádí vyhodnocení výrazu *výraz* (viz kapitola 5) a případné přiřazení jeho hodnoty do levého operandu *id*. Levý operand musí být proměnná (tzv. l-hodnota) a po přiřazení bude *id* stejného typu jako typ hodnoty výrazu *výraz*. Výsledkem příkazu je hodnota výrazu *výraz*. Část '*id* **=**' lze vynechat, takže hodnota vyhodnoceného výrazu *výraz* není nikam přiřazena.
- *Podmíněný příkaz*:
  if *výraz* **: EOL**
  sekvence_příkazů$_1$
  else : EOL
  sekvence_příkazů$_2$
  Sémantika příkazu je následující: Nejprve se vyhodnotí daný *výraz* (typicky využívající některý relační operátor). 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 relačního operátoru dle sekce 5.1) je považována hodnota **`None`**, **`''`** a **`0`** za nepravdu a ostatní hodnoty za pravdu. *Sekvence_příkazů*$_1$ a *sekvence_příkazů*$_2$ jsou neprázdné sekvence dílčích příkazů definované v této sekci (rekurzivní definice).
- *Příkaz cyklu*:
  while *výraz* **: EOL**
  sekvence_příkazů
  Sémantika příkazu cyklu je následující: Opakuje provádění sekvence příkazů *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 podmíněného příkazu.
- *Volání vestavěné či uživatelem definované funkce*:
  id **=** *název_funkce* **(***seznam_vstupních_parametrů***)**
  Seznam_vstupních_parametrů je seznam termů (viz sekce 3.1) oddělených čárkami[16]. Seznam může být i prázdný. Sémantika vestavěných funkcí bude popsána v kapitole 6. Sémantika volání funkce je následující: Příkaz zajistí předání parametrů hodnotou a předání řízení do těla funkce. V případě, že příkaz volání funkce obsahuje jiný počet parametrů, než funkce očekává (tedy než je uvedeno v její hlavičce, a to i u vestavěných funkcí), jedná se o chybu 5. Po návratu z těla funkce dojde k případnému uložení výsledku funkce do *id* a pokračování běhu programu bezprostředně za příkazem volání právě provedené funkce. Analogicky s příkazem přiřazení lze část '*id* **=**' vynechat.
- *Příkaz návratu z funkce*:
  return *výraz*
  Příkaz může být použit pouze v tělech funkcí (tj. ne v těle hlavního programu). Jeho sémantika je následující: Dojde k vyhodnocení výrazu *výraz* (tj. získání návratové hodnoty), okamžitému ukončení provádění těla funkce a návratu do místa volání,

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

kam funkce vrátí vypočtenou návratovou hodnotu. Není-li proveden žádný příkaz **`return`** před dokončením provádění funkce, vrací se implicitní hodnota **`None`**. Je-li výraz *výraz* vynechán, vrací se též hodnota **`None`**.

- *Prázdný příkaz*:

  `pass`

  Příkaz nic neprovádí a typicky slouží pro zápis prázdné sekvence dílčích příkazů.

5 Výrazy

Výrazy jsou tvořeny termy, závorkami a binárními aritmetickými, řetězcovým a relačními operátory.

Je-li to nutné, jsou prováděny implicitní konverze operandů i výsledků výrazů či funkcí. Jediná podporovaná implicitní konverze datových typů je z typu int na float.

Pro chybné kombinace datových typů (po případných povolených implicitních konverzích), které jste schopni ověřit při překladu[17], vracejte chybu 4. Ostatní typové kontroly generujte do mezikódu a při běhové chybě kvůli nekompatibilitě typů ukončete interpret s návratovým kódem 4.

5.1 Aritmetické, řetězcové a relační operátory

Standardní binární operátory **`+`**, **`-`**, **`*`** značí sčítání, odčítání[18] a násobení. Jsou-li oba operandy typu int, je i výsledek typu int. Je-li jeden[19] či oba operandy typu float, výsledek je též typu float. Operátor **`+`** navíc provádí se dvěma operandy typu str jejich konkatenaci.

Operátor **`/`** značí dělení dvou číselných operandů a výsledek je vždy typu float. Operátor **`//`** značí celočíselné dělení očekávající celočíselné operandy a dávající výsledek typu int. Při dělení nulou operátorem **`/`** nebo **`//`** dochází k běhové[20] chybě 9.

Pro relační operátory **`<`**, **`>`**, **`<=`**, **`>=`**, **`==`**, **`!=`** platí, že výsledkem porovnání je pravdivostní hodnota a že mají stejnou sémantiku jako v jazyce Python 3. Tyto operátory pracují s operandy stejného typu, a to int, float nebo str. Je-li jeden operand int a druhý float, je operand typu int konvertován na float. U řetězců se porovnání provádí lexikograficky. Operátory **`==`** a **`!=`** umožňují porovnávat i operandy různých typů (včetně **`None`**[21]). Nedojde-li u různých typů k implicitní konverzi z typu int na float, je pro operátor **`==`** výsledek nepravda (pro **`!=`** pravda). Bez rozšíření BOOLOP není s výsledkem porovnání možné dále pracovat a lze jej využít pouze u příkazů **`if`** a **`while`**.

5.2 Priorita operátorů

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

[17] Statická detekce typových chyb v konstantních výrazech může být různě propracovaná, takže hodnotící testy budou v těchto situacích uznávat chybu 4 i jako běhovou (vracenou z interpretu).

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

[19] Pak samozřejmě proběhne implicitní konverze druhého operandu též na float.

[20] Při rozpoznání dělení nulou jako konstantním výrazem je možné hlásit chybu 9 již při překladu.

[21] V jazyce Python 3 se preferuje pro testování hodnoty None operátor is, který však v IFJ19 nemáme.

Priorita	Operátory	Asociativita
1	`*` `/` `//`	levá
2	`+` `-`	levá
3	`<` `<=` `>` `>=` `==` `!=`	žádná[22]

6 Vestavěné funkce

Překladač bude poskytovat některé základní vestavěné funkce, které bude možné využít v programech jazyka IFJ19. Pro generování kódu vestavěných funkcí lze výhodně využít specializovaných instrukcí jazyka IFJcode19.

Vestavěné funkce pro načítání literálů a výpis termů:

- **`inputs()`**
  `inputi()`
  `inputf()`
  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 je známkou špatného formátu. 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 výsledkem hodnota **`None`**.
- *Příkaz pro výpis hodnot*:
  `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 dle typu v patřičném formátu. Výpis jednotlivých termů je oddělen mezerou a za posledním termem se vypisuje místo mezery odřádkování (znak s ASCII hodnotou 10). Hodnota termu typu int bude vytištěna pomocí `'%d'`[23], hodnota termu typu float pak pomocí `'%a'`[24]. **`None`** je vypsán jako řetězec `None`. Funkce **`print`** vždy vrací návratovou hodnotu **`None`**.

Vestavěné funkce pro práci s řetězci: V rámci následujících vestavěných funkcí probíhají i případné implicitní konverze parametrů a při špatném typu parametru dochází k chybě 4.

- **`len(`***s***`)`** – Vrátí délku (počet znaků) řetězce zadaného jediným parametrem *s*. Např. **`len("x\nz")`** vrací 3.
- **`substr(`***s***`,` *i*`,` *n*`)`** – Vrátí podřetězec zadaného řetězce *s*. Druhým parametrem *i* je dán začátek požadovaného podřetězce (počítáno od nuly) a třetí parametr *n* určuje délku

[22] Asociativitu relačních operátorů není v základu třeba implementovat, případně viz rozšíření BOOLOP.
[23] Formátovací řetězec standardní funkce **`printf`** jazyka C (standard C99 a novější).
[24] Formátovací řetězec **`printf`** jazyka C pro přesnou hexadecimální reprezentaci desetinného čísla.

podřetězce. Je-li index i mimo meze 0 až **`len(`**s**`)`** nebo $n < 0$, vrací funkce **`None`**. Je-li $n >$ **`len(`**s**`)`** $-i$, jsou jako řetězec vráceny od i-tého znaku všechny zbývající znaky řetězce s.

- **`ord(`**s**`, `** i **`)`** – Vrátí ordinální hodnotu (ASCII) znaku na pozici i v řetězci s. Je-li pozice mimo meze řetězce (0 až **`len(`**s**`)`** - 1), vrací **`None`**.
- **`chr(`**i**`)`** – Vrátí jednoznakový řetězec se znakem, jehož ASCII kód je zadán parametrem i. Případ, kdy je i mimo interval $[0; 255]$, vede na běhovou chybu při práci s řetězcem.

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 IFJ19.

8.1 Výpočet faktoriálu (iterativně)

```
# Program 1: Vypocet faktorialu (iterativne)
print('Zadejte cislo pro vypocet faktorialu: ')
a = inputi()
if a < 0:  # chybi kontrola a == None pro pripad nevalidniho vstupu
  print("""
Faktorial nelze spocitat
""")
else:
  vysl = 1
  while a > 0:
    vysl = vysl * a
    a = a - 1
  print('Vysledek je:', vysl, '\n')
```

8.2 Výpočet faktoriálu (rekurzivně)

```
# Program 2: Vypocet faktorialu (rekurzivne)
def factorial(n):
  if n < 2:
    result = 1
  else:
# else cast prikazu vetveni, jednoradkovy komentar, negeneruje DEDENT
```

```
    decremented_n = n - 1
    temp_result = factorial(decremented_n)
    result = n * temp_result
  return result
# end of factorial function

# Hlavni telo programu
print('Zadejte cislo pro vypocet faktorialu: ')
a = inputi()
if a < 0.0:  # pred porovnanim probehne implicitni konverze int na float
  print('Faktorial nelze spocitat')
else:
  vysl = factorial(a)
  print('Vysledek je:', vysl)
```

8.3 Práce s řetězci a vestavěnými funkcemi

```
""" Program 3: Prace s retezci a vestavenymi funkcemi """
s1 = 'Toto je nejaky text'
s2 = s1 + ', ktery jeste trochu obohatime'
print(s1, '\n', s2)
s1len = len(s1)
s1len = s1len - 4
s1 = substr(s2, s1len, 4)
s1len = s1len + 1  # korekce indexu znaku na poradi znaku
print('4 znaky od ', s1len, '. znaku v "', s2, '\":', s1)
print('Zadejte serazenou posloupnost vsech malych pismen a-h, ')
print('pricemz se pismena nesmeji v posloupnosti opakovat: ')
s1 = inputs()
if s1 != None:
  while s1 != 'abcdefgh':
    print('Spatne zadana posloupnost, zkuste znovu: ')
    s1 = inputs()
else:  # nezadan zadny vstup (EOF)
  pass
```

9 Doporučení k testování

Programovací jazyk IFJ19 je schválně navržen tak, aby byl téměř kompatibilní s podmnožinou jazyka Python 3[25]. Pokud si student není jistý, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ19, může si to ověřit následovně. Z IS FIT si stáhne ze *Souborů* k předmětu IFJ ze složky *Projekt* soubor `ifj19.py` obsahující kód, který doplňuje kompatibilitu IFJ19 s interpretem `python3` jazyka Python 3 na serveru `merlin`. Soubor `ifj19.py` obsahuje definice vestavěných funkcí, které jsou součástí jazyka IFJ19, ale chybí v základních knihovnách jazyka Python, nebo tyto redefinují.

Váš program v jazyce IFJ19 uložený například v souboru `testPrg.ifj` pak lze provést na serveru `merlin` například pomocí příkazu:

```
cat ./ifj19.py testPrg.ifj > testPrg.py
python3 testPrg.py < test.in > test.out
```

[25] Online dokumentace k Python 3: `https://docs.python.org/3/index.html`

Tím lze jednoduše zkontrolovat, co by měl provést zadaný zdrojový kód resp. vygenerovaný cílový kód. Je ale potřeba si uvědomit, že jazyk Python 3 je nadmnožinou jazyka IFJ19, a tudíž může zpracovat i konstrukce, které nejsou v IFJ19 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 IFJcode19

Cílový jazyk IFJcode19 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 IFJcode19 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode19 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode19
```

10.1 Hodnotící interpret `ic19int`

Pro hodnocení a testování mezikódu v IFJcode19 je k dispozici interpret pro příkazovou řádku (`ic19int`):

```
ic19int 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 IFJcode19.
- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode19.
- 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. IFJcode19 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 IFJcode19 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 IFJ19.

Zápis každé konstanty v IFJcode19 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 32-bitové celé číslo (rozsah C-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, 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 neplatnost 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 IFJ19.

WRITE ⟨*symb*⟩ Výpis hodnoty na standardní výstup

Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **`print`** jazyka IFJ19 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 IFJcode19. 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 návratovou chybou ⟨*symb*⟩, kde ⟨*symb*⟩ je celé číslo v intervalu 0 až 49 (včetně). Nevalidní celočíselná hodnota ⟨*symb*⟩ vede na chybu 57.

10.4.8 Ladící 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[26], 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 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.

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

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.

Dokumentace nesmí:

- obsahovat kopii zadání či text, obrázky[27] 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 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 něm 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 IFJcode19 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.

[27] Vyjma obyčejného loga fakulty na úvodní straně.

12.5 Jak postupovat při řešení projektu

Při řešení je pochopitelně možné využít vlastní výpočetní techniku. Instalace překladače `gcc` není nezbytně nutná, pokud máte jiný překladač jazyka C již instalován a nehodláte využívat vlastností, 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 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 IFJ19

Popis rozšíření vždy začíná jeho identifikátorem. Většina těchto rozšíření je založena na dalších vlastnostech jazyka Python 3. Podrobnější informace lze získat ze specifikace jazyka[2] Python 3. Do dokumentace je potřeba (kromě zkratky na úvodní stranu) také uvést, jak jsou implementovaná rozšíření řešena.

- BOOLOP: Podpora typu bool, booleovských hodnot **`True`** a **`False`**, booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**`not`**, **`and`**, **`or`**), jejichž priorita a asociativita odpovídá jazyku Python 3. Pravdivostní hodnoty lze porovnávat všemi relačními operátory. Dále podporujte výpisy hodnot typu bool, implicitní konverze na číselné typy a přiřazování výsledku booleovského výrazu do proměnné (+1,0 bodu). Navíc podporujte netradiční asociativitu relačních operátorů[28] pro operandy kompatibilních typů. Např. `a == b == c` se vyhodnocuje jako `a == b and b == c`.
- 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 velký písmena '**`B`**', '**`O`**' a '**`X`**'. Navíc lze mezi číslicemi použít znak podtržítko ('_') pro přehlednější zápis velkých čísel dle pravidel v jazyce Python verze 3.6.9.
- CYCLES: Překladač bude podporovat i cykly tvořené klíčovým slovem **`for`** pro iteraci nad znaky zadaného řetězce. Dále bude podporovat v obou typech iterace klíčová slova **`break`** a **`continue`** a také klauzuli **`else`** pro případ, že není tělo cyklu provedeno ani jednou (+1,0 bodu).
- FUNEXP: Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce. Na rozdíl od základního zadání je třeba podporovat **`EOL`**

[28] Viz `https://docs.python.org/3/reference/expressions.html#comparisons`.

mezi závorkami (tj. mezi parametry funkcí a ve výrazech obklopených závorkami), kdy stejně jako v Python 3 ignorujte nekonzistence odsazení až do odpovídající uzavírající závorky (+1,5 bodu).

- IFTHEN: Podporujte zjednodušený podmíněný příkaz **`if`** bez části **`else`**, rozšířený podmíněný příkaz s volitelným vícenásobným výskytem **`elif`** části a ve výrazech podporujte ternární operátor **`if-else`** (+1,0 bodu).