ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL

Zbyněk Křivka, Ondřej Ondryáš, Radovan Klembara, Radim Kocman
email: {krivka, iondryas, iklembara, kocman}@fit.vut.cz
25. září 2024

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka IFJ24.
Informace: diskuzní fórum a Moodle předmětu IFJ.
Pokusné odevzdání: čtvrtek 21. listopadu 2024, 23:59 (nepovinné).
Datum odevzdání: středa 4. prosince 2024, 23:59.
Způsob odevzdání: prostřednictvím StudIS, aktivita „Projekt - Registrace a Odevzdání“.

Hodnocení:

- Do předmětu IFJ získá každý maximálně 28 bodů (18 celková funkčnost projektu (tzv. programová část), 5 dokumentace, 5 obhajoba).
- Do předmětu IAL získá každý maximálně 15 bodů (5 celková funkčnost projektu, 5 obhajoba, 5 dokumentace).
- 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 18 bodů zapsány do termínu „Projekt - Bonusové hodnocení“ v IFJ.

Řešitelské týmy:

- Projekt budou řešit čtyřčlenné týmy. Týmy s jiným počtem členů jsou nepřípustné (jen výjimečně tříčlenné).
- Vytváření týmů se provádí funkcionalitou Týmy v IS VUT. Tým vytváří vedoucí a název týmu bude automaticky generován na základě loginu vedoucího. Vedoucí má kontrolu nad složením týmu, smí předat vedení týmu jinému členovi a bude odevzdávat

výsledný archiv. Rovněž vzájemná komunikace mezi vyučujícími a týmy bude probíhat nejlépe prostřednictvím vedoucích (ideálně v kopii dalším členům týmu).

- Pokud bude mít tým dostatek členů, bude mu umožněno si zaregistrovat jednu ze dvou variant zadání (viz sekce 7) v aktivitě „Projekt - Registrace a Odevzdání“.
- Všechny hodnocené aktivity k projektu najdete ve StudIS, studijní informace v Moodle předmětu IFJ a další informace na stránkách předmětu[1].

2 Zadání

Vytvořte program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ24 a přeloží jej do cílového jazyka IFJcode24 (mezikód). Jestliže proběhne překlad bez chyb, vrací se návratová hodnota `0` (nula). Jestliže došlo k nějaké chybě, vrací se návratová hodnota následovně:

- 1 - chyba v programu v rámci lexikální analýzy (chybná struktura aktuálního lexému).
- 2 - chyba v programu v rámci syntaktické analýzy (chybná syntaxe programu, chybějící hlavička, atp.).
- 3 - sémantická chyba v programu – nedefinovaná funkce či proměnná.
- 4 - sémantická chyba v programu – špatný počet/typ parametrů u volání funkce; špatný typ či nepovolené zahození návratové hodnoty z funkce.
- 5 - sémantická chyba v programu – redefinice proměnné nebo funkce; přiřazení do nemodifikovatelné proměnné.
- 6 - sémantická chyba v programu – chybějící/přebývající výraz v příkazu návratu z funkce.
- 7 - sémantická chyba typové kompatibility v aritmetických, řetězcových a relačních výrazech; nekompatibilní typ výrazu (např. při přiřazení).
- 8 - sémantická chyba odvození typu – typ proměnné není uveden a nelze odvodit od použitého výrazu.
- 9 - sémantická chyba nevyužité proměnné v jejím rozsahu platnosti; modifikovatelná proměnná bez možnosti změny po její inicializaci.
- 10 - ostatní sémantické chyby.
- 99 - interní chyba překladače tj. neovlivněná vstupním programem (např. chyba alokace paměti atd.).

Překladač bude načítat řídicí program v jazyce IFJ24 ze standardního vstupu a generovat výsledný mezikód v jazyce IFJcode24 (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 bez grafického uživatelského rozhraní (tzv. filtr). Pro interpretaci výsledného programu v cílovém jazyce IFJcode24 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.vut.cz/study/courses/IFJ/public/project`

3 Popis programovacího jazyka

Jazyk IFJ24 je zjednodušenou podmnožinou jazyka Zig[2] (ve verzi 0.13.0), což je moderní a silně typovaný jazyk navržený jako alternativa jazyka C, jehož cílem je zjednodušit vývoj robustního, znovupoužitelného, udržovatelného a bezpečného software.

3.1 Obecné vlastnosti a datové typy

V programovacím jazyce IFJ24 **záleží** na velikosti písmen u identifikátorů i klíčových slov (tzv. *case-sensitive*). Jazyk IFJ24 je silně typovaný s podporou jednoduchého odvozování typů proměnných.

- *Identifikátor* je definován jako neprázdná posloupnost číslic, písmen (malých i velkých) a znaku podtržítka ('_') začínající písmenem nebo podtržítkem (kromě samotného podtržítka). Jazyk IFJ24 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam[3], a proto se nesmějí vyskytovat jako identifikátory:

  Klíčová slova: **`const`**, **`else`**, **`fn`**, **`if`**, **`i32`**, **`f64`**, **`null`**, **`pub`**, **`return`**, **`u8`**, **`var`**, **`void`**, **`while`**

- *Identifikátor vestavěné funkce IFJ24* je definován jako tečkou oddělený jmenný prostor '**`ifj`**' a identifikátor konkrétní funkce. Mezi jednotlivými částmi se může vyskytovat libovolný počet bílých znaků. Např. **`ifj . write`**.
- *Identifikátor typu* se skládá z volitelné předpony, což je znak '**`?`**' (otazník), a klíčového slova pro typ (**`f64`**, **`i32`**) nebo složitější označení typu pro *řez* pole znaků **`[]u8`** (angl. *slice*). Např. **`i32`** nebo **`?[]u8`**.
- *Celočíselný literál* typu **`i32`** (rozsah int v IFJcode24) je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě. Nenulový literál nesmí začínat znakem 0.
- *Desetinný literál* typu **`f64`** (rozsah float v IFJcode24) také vyjadřuje nezáporná čísla v desítkové soustavě, přičemž literál je tvořen celou a desetinnou částí, nebo celou částí a exponentem, nebo celou a desetinnou částí a exponentem. Celá i desetinná část je tvořena neprázdnou posloupností číslic. Exponent je celočíselný, začíná znakem '**`e`**' nebo '**`E`**', následuje nepovinné znaménko '**`+`**' (plus) nebo '**`-`**' (mínus) a poslední částí je neprázdná posloupnost číslic. Mezi jednotlivými částmi nesmí být jiný znak, celou a desetinnou část odděluje znak '`.`' (tečka)[4].
- *Řetězcový literál*[5] je oboustranně ohraničen dvojitými uvozovkami (`"`, ASCII hodnota 34). Tvoří jej libovolný počet znaků zapsaných na jednom řádku (nemůže obsahovat odřádkování). Možný je i prázdný řetězec (**`""`**). Znaky s ASCII hodnotou větší než

[2] Online dokumentace: `https://ziglang.org/documentation/0.13.0/`; na serveru Merlin pod příkazem `zig` je pro studenty k dispozici verze 0.13.0 z 7. 6. 2024.

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

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

[5] Označení typu pro řetězcový literál není do jazyka IFJ24 zahrnuto.

31 (mimo `"`) lze zapisovat přímo. Některé další znaky lze zapisovat pomocí escape sekvence: '`\"`', '`\n`', '`\r`', '`\t`', '`\\`'. Jejich význam se shoduje s odpovídajícími znakovými konstantami jazyka Zig[6]. Pokud znaky za zpětným lomítkem neodpovídají žádnému z uvedených vzorů[7], dojde k chybě 1. Znak v řetězci může být zadán také pomocí escape sekvence '`\x`*dd*', kde *dd* je hexadecimální číslo složeno z právě dvou hexadecimálních číslic (písmena `A-F` mohou být malá i velká).
Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

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

reprezentuje řetězec
`Ahoj`
`"Sve'te \"`. Neuvažujte řetězce, které obsahují vícebajtové znaky kódování Unicode (např. UTF-8).
Podporujte také víceřádkové řetězce, které začínají '`\\`' a pokračují až do konce řádku (není zahrnut). Řetězec pokračuje na další řádek, pokud tento řádek začíná (bílé znaky jsou ignorovány) dvojicí '`\\`' – v takovém případě je v řetězci zahrnut znak konce řádku '`\n`'.

- *Datové typy* jsou `i32`, `f64` a `[]u8`. Řetězcový literál je nutné na řez pole znaků (typ `[]u8`) převést vestavěnou funkcí `ifj.string()`. Typy se deklarují u proměnných a také v signaturách funkcí. Chybějící deklarace typu proměnné bude překladačem odvozena z inicializačního výrazu. Speciální hodnotu `null` mohou nabývat všechny proměnné, parametry a návratové hodnoty typované s předponou '`?`', tzv. *typ zahrnující* ***`null`***. Z hodnoty `null` nelze odvodit konkrétní typ.
- *Term* je libovolný literál (celočíselný, desetinný, řetězcový či `null`) nebo identifikátor.
- Jazyk IFJ24 podporuje pouze *řádkové* komentáře. Řádkový komentář začíná dvojicí lomítek ('`//`', ASCII hodnota 47) a za komentář je považováno vše, co následuje až do konce řádku.

4 Struktura jazyka

IFJ24 je strukturovaný programovací jazyk podporující definice modifikovatelných a nemodifikovatelných proměnných a uživatelských funkcí včetně jejich rekurzivního volání.

4.1 Základní struktura jazyka

Program se skládá z prologu následovaného sekvencí definic uživatelských funkcí včetně definice *hlavní funkce* `main`, která je povinným vstupním bodem programu. Chybějící funkce `main` způsobí chybu 3. Funkce `main` nemá definován žádný parametr ani návratovou hodnotu, jinak chyba 4.

[6] `https://ziglang.org/documentation/0.13.0/#Escape-Sequences`
[7] A nejde ani o escape sekvenci podporovanou jazykem Zig.

Prolog[8] se skládá z jednoho řádku:

```
const ifj = @import("ifj24.zig");
```

V tělech funkcí lze potom definovat proměnné, používat příkazy přiřazení, větvení, iterace a volání funkcí. V těle uživatelské funkce lze navíc vracet její výsledek.

Před, za i mezi jednotlivými tokeny se může vyskytovat libovolný počet bílých znaků (mezera, tabulátor, komentář a odřádkování), takže jednotlivé konstrukce jazyka IFJ24 lze zapisovat na jednom či více řádcích. Za každým základním příkazem se píše znak '**;**' (středník), naopak za příkazem končícím blokem, tj. zavírací složenou závorkou, se středník uvádět nesmí a nelze uvést ani středník bez příkazu (tzv. prázdný příkaz). Na jednom řádku lze zapsat více příkazů.

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

4.1.1 Proměnné

Proměnné jazyka IFJ24 jsou pouze lokální. Lokální proměnné jsou definované v uživatelských funkcích a jejich podblocích. Lokální proměnné mají rozsah platnosti od místa jejich definice až po konec bloku, ve kterém byly definovány. *Blokem* je libovolná sekvence příkazů zapsána v těle funkce, v rámci větve podmíněného příkazu nebo příkazu cyklu. Blok a jeho podbloky tvoří tzv. *rozsah platnosti*, kde je proměnná dostupná. V podblocích nelze žádnou již definovanou proměnnou překrýt (angl. *shadowing*), takže redefinice proměnné v témže bloku či podbloku vede na chybu 5.

Speciální pseudoproměnná _ se používá pro zahození (deklarované nevyužití) výsledku výrazu či volání funkce s návratovou hodnotou.

Definice nemodifikovatelné proměnné se provádí příkazem **`const`** a modifikovatelné (běžné) proměnné příkazem **`var`**. V obou případech musí za **=** následovat inicializační výraz. Detailní syntaxe bude popsána v sekci 4.3. Každá proměnná musí být definována před jejím použitím, což musí být staticky analyzovatelné, jinak se jedná o chybu 3. Analogicky každá definovaná proměnná musí být v jejím rozsahu platnosti využita, jinak chyba 9.

4.2 Deklarace a definice uživatelských funkcí

Definice funkce se skládá z hlavičky a těla funkce. Definice funkce je zároveň její deklarací. Každá použitá funkce musí být definovaná, jinak končí analýza chybou 3. Přetěžování funkcí (angl. *overloading*) je v IFJ24 zakázáno.

Definice funkce nemusí lexikálně předcházet kódu pro použití této funkce, tzv. *volání funkce* (příkaz volání je definován v sekci 4.3). Uvažujte například vzájemné rekurzivní volání funkcí (tj. funkce `foo` volá funkci `bar`, která opět může volat funkci `foo`).

Příklad (vzájemná rekurze bez ukončující podmínky):

```
const ifj = @import("ifj24.zig");
pub fn bar(param : []u8) []u8 {
    const r = foo(param);
    return r;
```

[8] Prolog mohou prokládat komentáře a prázdné řádky a slouží především k zajištění kompatibility s programy jazyka Zig.

```
}
pub fn foo(par : []u8) []u8 {
    const ret = bar(par);
    return ret;
}

pub fn main() void {
        const par = ifj.string("ahoj");
        _ = bar(par);
}
```

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

pub fn *id* **(** *seznam_parametrů* **)** *návratový_typ* **{**
sekvence_příkazů
}

- Hlavička definice funkce sahá od klíčových slov **pub fn** až po určení návratového typu funkce, pak následuje tělo funkce tvořené sekvencí příkazů (viz sekce 4.3) ve složených závorkách.
- Seznam parametrů je tvořen posloupností definic parametrů oddělených čárkou, přičemž za posledním parametrem se čárka může, ale nemusí uvádět. Seznam může být i prázdný. Každá definice parametru obsahuje identifikátor a typ parametru:

  identifikátor_parametru : *typ*

  Identifikátor parametru slouží jako identifikátor v těle funkce pro získání hodnoty tohoto parametru. Parametry jsou vždy předávány hodnotou. Příklad:

  ```
  const ifj = @import("ifj24.zig");

  pub fn build(x : []u8, y : []u8) []u8 {
    const res = ifj. concat(x, y);
    return res;
  }

  pub fn main() void {
    const a = ifj.string("ahoj ");
    var ct : []u8 = ifj.string("svete");
    ct = build(a, ct);
    ifj.write(ct);
  }
  ```

- V těle funkce jsou její parametry chápány jako předdefinované lokální proměnné s implicitní hodnotou danou skutečnými parametry, které nelze v těle funkce měnit. Výsledek funkce je dán provedeným příkazem návratu z funkce (viz sekce 4.3).

Každá funkce s návratovou hodnotou vrací hodnotu výrazu z příkazu **return**, avšak v případě chybějící návratové hodnoty kvůli neprovedení žádného příkazu **return** nastane chyba 6, nebo provedením příkazu **return**, ale s výrazem špatného typu (typ návratové hodnoty neodpovídá návratovému typu funkce), dochází k chybě 4.

Funkce bez návratové hodnoty (návratový typ v hlavičce je **void**, tedy tzv. *void-funkce*) je ukončena po posledním příkazu těla funkce nebo provedením příkazu **return** bez výrazu pro návratovou hodnotu. V IFJ24 nelze definovat vnořené funkce.

Struktura jednotlivých příkazů je popsána v následující sekci.

4.3 Syntaxe a sémantika příkazů

Sekvence příkazů (i prázdná) je uzavřená ve složených závorkách a tvoří nový *blok*, který má vlastní rozsah platnosti proměnných v něm definovaných.
Dílčím příkazem se rozumí:

- *Příkaz definice proměnné*:
  `const` *id* **`:`** *typ* **`=`** *výraz* **`;`**
  `var` *id* **`:`** *typ* **`=`** *výraz* **`;`**
  Sémantika příkazu je následující: Příkaz definuje lokální proměnnou, která v daném či žádném nadřazeném bloku ještě nebyla definována, jinak dojde k chybě 5. Příkaz provádí vyhodnocení povinného inicializačního *výrazu* (viz kapitola 5) a přiřazení jeho hodnoty do proměnné *id*. Levý operand *id* je pro klíčové slovo **`var`** proměnná (tzv. l-hodnota) nebo pro klíčové slovo **`const`** nemodifikovatelná proměnná. Hodnotu nemodifikovatelné proměnné nelze po inicializaci již dále měnit. Je-li možné při překladu odvodit typ výrazu přiřazovaného do proměnné, tak je možné část '**`: typ`**' vypustit a typ proměnné *id* odvodit. Chybí-li *výraz*, jedná se o syntaktickou chybu. Nelze-li chybějící typ odvodit (např. z hodnoty **`null`**), dojde k chybě 8. Není-li typ zapsaného *výrazu* kompatibilní s uvedeným typem *typ*, nastane chyba 7.
- *Příkaz přiřazení*:
  id **`=`** *výraz* **`;`**
  Sémantika příkazu je následující: Příkaz provádí vyhodnocení výrazu *výraz* (viz kapitola 5) a přiřazení jeho hodnoty do levého operandu *id*, který je modifikovatelnou proměnnou (tj. dříve definovanou pomocí klíčového slova **`var`**). Pokud není hodnota výrazu typově kompatibilní s typem proměnné *id*, dojde k chybě 7.
  V případě, že se přiřazuje řetězcový literál, tak je potřeba využít funkci **`ifj.string`**, aby byl tento literál transformován na řez (odpovídající **`[]u8`**), jinak je hodnota[9] takové proměnné nevyužitelná, což vede na chybu 7 nebo při inicializaci na chybu 8. Po transformaci na řez lze tento předávat funkcím jako parametr (viz příklad v sekci 8.3).
- *Podmíněný příkaz*:
  `if (`*pravdivostní_výraz***`)`**
  `{`
  sekvence_příkazů$_1$
  `}`
  `else`
  `{`
  sekvence_příkazů$_2$
  `}`
  Sémantika příkazu je následující: Nejprve se vyhodnotí *pravdivostní_výraz*, který je pravdivostního typu. Pokud je vyhodnocený výraz pravdivý, vykoná se *sekvence_příkazů*$_1$, jinak se vykoná *sekvence_příkazů*$_2$. Pokud výsledná hodnota výrazu není prav-

[9] V Zig je řetězcový literál typu **`[_]u8`**, což není kompatibilní s **`[]u8`**.

divostní (tj. pravda či nepravda – v základním zadání pouze jako výsledek aplikace relačních operátorů dle sekce 5.1), tak dojde k chybě 7.

- *Příkaz podmíněný neprázdnou hodnotou*:

  if (*výraz_s_null***) |***id_bez_null***|**
  {
  sekvence_příkazů$_1$
  }
  else
  {
  sekvence_příkazů$_2$
  }

  Nechť *výraz_s_null* je libovolný výraz typu zahrnujícího **null** (s předponou '**?**'). Pokud je jeho hodnota neprázdná (tj. různá od **null**), převede se výsledek výrazu na odpovídající hodnotu typu nezahrnujícího **null**, která je přiřazena do proměnné *id_bez_null* s rozsahem platnosti v bloku *sekvence_příkazů*$_1$. Tento blok se poté provede. V opačném případě se provede blok *sekvence_příkazů*$_2$ (zde *id_bez_null* není definována).

- *Příkaz cyklu*:

  while (*výraz***)**
  {
  sekvence_příkazů
  }

  Příkaz cyklu se skládá z hlavičky a těla tvořeného *sekvencí_příkazů*.
  Sémantika příkazu cyklu je následující: Opakuje provádění sekvence dílčích příkazů tak dlouho, dokud je hodnota výrazu pravdivá. Pravidla pro určení pravdivosti *výrazu* jsou stejná jako u pravdivostního výrazu v podmíněném příkazu.
  Analogicky k podmíněnému příkazu lze u příkazu cyklu mít hlavičku tvaru:

  while (*výraz_s_null***) |***id_bez_null***|**

  Sekvence příkazů se opakuje tak dlouho, dokud je hodnota *výraz_s_null* různá od **null**. V takovém případě se převede výsledek výrazu na odpovídající hodnotu typu nezahrnujícího **null**, která je přiřazena do proměnné *id_bez_null* s rozsahem platnosti v bloku *sekvence_příkazů*.

- *Volání vestavěné či uživatelem definované funkce*:

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

  Seznam_vstupních_parametrů je seznam termů (viz sekce 3.1) oddělených čárkami[10], kdy za posledním termem může být čárka. Seznam vstupních parametrů může být i prázdný. Sémantika vestavěných funkcí bude popsána v kapitole 6.
  Sémantika volání uživatelem definovaných funkcí je následující: Příkaz zajistí předání parametrů hodnotou a předání řízení do těla funkce. V případě, že příkaz volání funkce obsahuje jiný počet nebo typy skutečných parametrů, než funkce očekává (tedy než je uvedeno v její hlavičce, a to i u vestavěných funkcí, včetně případného předání **null**, kde to není očekáváno), jedná se o chybu 4. Po dokončení provádění

[10]Parametrem volání funkce není v základním zadání výraz. Podporu výrazů je možné doplnit v rámci nepovinného bodovaného rozšíření FUNEXP, viz sekce 12.7.1.

zavolané funkce je přiřazena návratová hodnota do proměnné *id* a běh programu pokračuje bezprostředně za příkazem volání právě provedené funkce. Neobsahuje-li tělo funkce příkaz **`return`** a funkce by měla vracet hodnotu, dojde k chybě 6. Je-li volána *void*-funkce, tak při pokusu o přiřazení výsledku do *id* dojde k chybě 7.
V případě, že před *id* zapíšeme klíčové slovo **`const`** nebo **`var`** a případně za *id* volitelně určení typu, dojde před přiřazením hodnoty výsledku zavolané funkce nejprve k definici nové proměnné *id*.
Nelze volat funkci vracející hodnotu (ani **`null`**) bez jejího přiřazení do proměnné (v takovém případě dojde k chybě 4). Pokud hodnota nemá být dále využita, tak pro prevenci chyby 9 nahraďte *id* pseudoproměnnou '_' (podtržítko), což explicitně značí, že má být vrácená hodnota zahozena.

- *Volání funkce bez navracení hodnoty (tj. void*-funkce*)*:
  název_funkce **`(`***seznam_vstupních_parametrů***`) ;`**
  Při volání *void*-funkce jsou *seznam_vstupních_parametrů* a sémantika příkazu analogické předchozímu příkazu. *void*-funkce nemusí obsahovat příkaz **`return`**, který musí být případně bez výrazu, jinak nastane chyba 6.
- *Příkaz návratu z funkce*:
  `return` *výraz* **`;`**
  Příkaz může být použit v těle libovolné funkce (i *main*). Jeho sémantika je následující: Dojde k vyhodnocení výrazu *výraz* (tj. získání návratové hodnoty), okamžitému ukončení provádění těla funkce a návratu do místa volání, kam funkce vrátí vypočtenou návratovou hodnotu. V případě těla *void*-funkce (např. *main*) musí být výraz vynechán, jinak nastane chyba 6. Uživatelem definovaná *void*-funkce nemusí příkaz **`return`** vůbec obsahovat. Chybí-li ve funkci vracející hodnotu výraz *výraz* u příkazu **`return`**, vede to též na chybu 6. Pokud funkce s návratovou hodnotou vrací návratovou hodnotu neočekávaného typu dle její definice, dochází k chybě 4.

5 Výrazy

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

V IFJ24 jsou v rámci sémantické analýzy vstupního programu prováděny silné typové kontroly (tj. s minimem implicitních typových konverzí). Ve výrazech si operátory kontrolují kompatibilitu operandů z hlediska jejich typů a případně i typu výsledku.

5.1 Aritmetické a relační operátory

Standardní binární operátory **`+`**, **`-`**, **`*`** značí sčítání, odčítání[11] a násobení. Jsou-li oba operandy typu **`i32`**, je i výsledek typu **`i32`**. Jsou-li oba operandy typu **`f64`**, výsledek je typu **`f64`**. Je-li jeden operand typu **`i32`** a druhý **`f64`** a celočíselný operand je zároveň literál, dojde k implicitní konverzi literálu[12] na typ **`f64`**. Operátor **`/`** značí dělení (desetinné pro desetinné operandy, celočíselné[13] pro celočíselné operandy).

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

[12] Na proměnné se v IFJ24 ani v Zig implicitní konverze neaplikují.

[13] V Zig operátor **`/`** nelze použít na záporná celá čísla vč. typu **`i32`**. V IFJ24 bude operátor celočíselného dělení odpovídat funkci **`@divFloor`** ze Zig zaokrouhlující výsledek k $-\infty$.

Konstantní výraz typu `f64` s nulovou desetinnou částí lze implicitně konvertovat na `i32`. Pro provedení explicitního přetypování z `f64` na `i32` lze použít vestavěnou funkci `ifj.f2i`, naopak pak `ifj.i2f` (viz kapitola 6).

Pro operátor `==` platí: Lze porovnávat proměnné stejných číselných typů a případně literály a konstanty různých číselných typů (spolu s bezeztrátovou implicitní konverzí na typ `f64`), pokud má alespoň jeden operand hodnotu známou při překladu, jinak dochází k chybě 7. Pokud jsou operandy stejného typu, tak se porovnají hodnoty daných operandů. S `null` je možné porovnat právě každý výraz typu zahrnujícího `null`[14]. Porovnávat lze také výrazy, jejichž typ se liší pouze v zahrnutí `null`.

Operátor `!=` je negací operátoru `==`.

Pro relační operátory `<`, `>`, `<=`, `>=` platí: Sémantika operátorů odpovídá jazyku Zig. Ani jeden operand nesmí být výraz typu zahrnujícího `null`. Porovnání je také možné, pokud lze provést implicitní konverzi literálů tak, aby nedošlo k újmě na přesnosti u konkrétního operandu (např. v `f > 2`, kde `f` je proměnná typu `f64`, je `2` bezeztrátově při překladu zkonvertováno na `2.0`; naopak porovnání `n > 1.5` způsobí chybu 7, pokud je `n` proměnná typu `i32`, jejíž hodnotu neznáme při překladu). Všechny typové nekompatibility ve výrazech jsou v IFJ24 sémantická chyba 7.

Žádný z operátorů nelze aplikovat na řezy[15] nebo řetězcové literály. Bez rozšíření BOOLTHEN není s výsledkem porovnání (pravdivostní hodnota) možné dále pracovat a lze jej využít pouze u podmínek příkazů `if` a `while`.

5.2 Priorita operátorů

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

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

6 Vestavěné funkce

Překladač bude poskytovat některé základní vestavěné funkce, které bude možné využít v programech jazyka IFJ24. Pro generování kódu vestavěných funkcí lze výhodně využít specializovaných instrukcí jazyka IFJcode24. Při použití špatného typu termu v parametrech následujících vestavěných funkcí dochází k chybě 4. Všechny vestavěné funkce jsou zařazeny do jmenného prostoru `ifj`.

[14] Výraz `null == null` je také pravdivý.

[15] Výjimkou jsou výrazy typu `?[]u8`, které lze porovnat s `null`.

Vestavěné funkce pro načítání literálů a výpis termů:

- *Funkce pro načítání hodnot*:
  pub fn ifj.readstr() ?[]u8
  pub fn ifj.readi32() ?i32
  pub fn ifj.readf64() ?f64
  Vestavěné funkce ze standardního vstupu načtou jeden řádek ukončený odřádkováním nebo koncem souboru (EOF). Funkce **readstr** tento řetězec vrátí bez symbolu konce řádku (načítaný řetězec nepodporuje escape sekvence). V případě **readi32** a **readf64** jsou jakékoli nevhodné (včetně okolních bílých) znaky známkou špatného formátu, což vede na návratovou hodnotu **null**. Funkce **readi32** načítá a vrací celé číslo, **readf64** desetinné číslo. V případě chybějící hodnoty na vstupu (např. načtení EOF) nebo jejího špatného formátu je vrácena hodnota **null**.
- *Funkce pro výpis hodnoty*:
  pub fn ifj.write(*term***) void**
  Sémantika je následující: Vypíše hodnotu termu *term* na standardní výstup ihned a bez žádných oddělovačů dle typu v patřičném formátu. Hodnota termu typu **i32** bude vytištěna pomocí `'%d'`[16], hodnota termu typu **f64** pak pomocí `'%a'`[17]. Hodnota **null** je tištěna jako literál **"null"**. Funkce **ifj.write** nemá návratovou hodnotu.

Vestavěné funkce pro konverzi číselných typů:

- **pub fn ifj.i2f(***term* **: i32) f64** – Vrátí hodnotu celočíselného termu *term* konvertovanou na desetinné číslo. Pro konverzi z celého čísla využijte odpovídající instrukci z IFJcode24.
- **pub fn ifj.f2i(***term* **: f64) i32** – Vrátí hodnotu desetinného termu *term* konvertovanou na celé číslo, a to oříznutím desetinné části. Pro konverzi z desetinného čísla využijte odpovídající instrukci z IFJcode24.

Vestavěné funkce pro práci s řezy:

Následující funkce pro práci s řetězci očekávají místo řetězcového literálu tzv. *řez* pole znaků (dynamická reprezentace řetězce uložitelná do proměnné), který lze vytvořit pomocí vestavěné funkce

pub fn ifj.string(*term***) []u8**
kde *term* je řetězcový literál nebo řez.

- **pub fn ifj.length(***s* **: []u8) i32** – Vrátí délku (počet znaků) řezu *s*.
- **pub fn ifj.concat(***s*1 **: []u8,** *s*2 **: []u8) []u8** – Vrátí nový řez, který vznikne spojením řezu *s*1 a *s*2 (konkatenace).
- **pub fn ifj.substring(***s* **: []u8,** *i* **: i32,** *j* **: i32) ?[]u8** – Vrátí podřez zadaného řezu *s*. Druhým parametrem *i* je dán index začátku požadovaného podřetězce (počítáno od nuly) a třetím parametrem *j* určuje index za posledním znakem podřetězce (též počítáno od nuly).
  Funkce dále vrací hodnotu **null**, nastane-li některý z těchto případů:

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

- $i < 0$
- $j < 0$
- $i > j$
- $i \geq$ **`ifj.length`**(s)
- $j >$ **`ifj.length`**(s)

- **`pub fn ifj.strcmp(`**$s1$ **`: []u8,`** $s2$ **`: []u8) i32`** – Analogicky jako stejnojmenná funkce standardní knihovny jazyka C vrací výsledek lexikografického porovnání (-1 pro $s1$ menší než $s2$, 0 pro rovnost, 1 pro $s1$ větší než $s2$) dvou řezů reprezentujících řetězce v IFJ24.
- **`pub fn ifj.ord(`**s **`: []u8,`** i **`: i32) i32`** – Vrátí ordinální hodnotu (ASCII) i-tého znaku v řezu s (indexováno od nuly). Je-li řez prázdný nebo index mimo meze řezu, vrací funkce hodnotu **`0`**.
- **`pub fn ifj.chr(`**i **`: i32) []u8`** – Vrátí jednoznakový řez se znakem, jehož ASCII kód je zadán parametrem i. Hodnotu i mimo interval $[0; 255]$ řeší odpovídající instrukce IFJcode24.

7 Implementace tabulky symbolů

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

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

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

Implementace tabulky symbolů bude uložena v souboru `symtable.c` (případně `symtable.h`). Více viz sekce 12.2.

8 Příklady

Tato kapitola uvádí tři jednoduché příklady řídicích programů v jazyce IFJ24.

8.1 Výpočet faktoriálu (iterativně)

```
// Program 1: Vypocet faktorialu (iterativne)
const ifj = @import("ifj24.zig");
// Hlavni telo programu - funkce main
pub fn main() void {
    ifj.write("Zadejte cislo pro vypocet faktorialu\n");
    const a = ifj.readi32();
    if (a) |val| {
        if (val < 0) {
            ifj.write("Faktorial "); ifj.write(val);
```

```
            ifj.write(" nelze spocitat\n");
        } else {
            var d: f64 = ifj.i2f(val);
            var vysl: f64 = 1.0;
            while (d > 0) {
                vysl = vysl * d;
                d = d - 1.0;
            }
            ifj.write("Vysledek: "); ifj.write(vysl); ifj.write(" = ");
            const vysl_i32 = ifj.f2i(vysl);
            ifj.write(vysl_i32); ifj.write("\n");
        }
    } else { // a == null
        ifj.write("Faktorial pro null nelze spocitat\n");
    }
}
```

8.2 Výpočet faktoriálu (rekurzivně)

```
// Program 2: Vypocet faktorialu (rekurzivne)
const ifj = @import("ifj24.zig");
// Hlavni funkce
pub fn main() void {
    ifj.write("Zadejte cislo pro vypocet faktorialu: ");
    const inp = ifj.readi32();
    if (inp) |INP| {
        if (INP < 0) {
            ifj.write("Faktorial nelze spocitat!\n");
        } else {
            const vysl = factorial(INP);
            ifj.write("Vysledek: "); ifj.write(vysl);
        }
    } else {
        ifj.write("Chyba pri nacitani celeho cisla!\n");
    }
}
// Pomocna funkce pro dekrementaci celeho cisla o zadane cislo
pub fn decrement(n: i32, m: i32) i32 {
    return n - m;
}
// Definice funkce pro vypocet hodnoty faktorialu
pub fn factorial(n: i32) i32 {
    var result: i32 = 0 - 1;
    if (n < 2) {
        result = 1;
    } else {
        const decremented_n = decrement(n, 1);
        const temp_result = factorial(decremented_n);
        result = n * temp_result;
    }
    return result;
}
```

8.3 Práce s řetězci a vestavěnými funkcemi

```
// Program 3: Prace s retezci a vestavenymi funkcemi
const ifj = @import("ifj24.zig");

// Hlavni funkce
pub fn main() void {
    const str1 = ifj.string("Toto je text v programu jazyka IFJ24");
    var str2 = ifj.string(", ktery jeste trochu obohatime");
    str2 = ifj.concat(str1, str2);
    ifj. write(str1); ifj . write("\n");
    ifj .write(str2); ifj.write("\n");
    ifj.write("Zadejte serazenou posloupnost malych pismen a-h:\n");
    var newInput = ifj.readstr();
    var all: []u8 = ifj.string("");
    while (newInput) |inpOK| {
        const abcdefgh = ifj.string("abcdefgh");
        const strcmpResult = ifj.strcmp(inpOK, abcdefgh);
        if (strcmpResult == 0) {
            ifj.write("Spravne zadano!\n");
            ifj.write(all); // vypsat spojene nepodarene vstupy
            newInput = null; // misto break;
        } else {
            ifj.write("Spatne zadana posloupnost, zkuste znovu:\n");
            all = ifj.concat(all, inpOK); // spojuji neplatne vstupy
            newInput = ifj.readstr();
        }
    }
}
```

9 Doporučení k testování

Programovací jazyk IFJ24 je schválně navržen tak, aby byl téměř kompatibilní s podmnožinou jazyka Zig[18]. Pokud si nejste jistí, co by měl cílový kód přesně vykonat pro nějaký zdrojový kód jazyka IFJ24, můžete si to ověřit následovně. Z Moodle si stáhnete ze souborů k projektu soubor `ifj24.zig` obsahující kód, který doplňuje kompatibilitu IFJ24 s překladačem `zig` jazyka Zig na serveru `merlin`. Soubor `ifj24.zig` obsahuje definice vestavěných funkcí, které jsou prologem importovány do jmenného prostoru **`ifj`**. Váš program v jazyce IFJ24 uložený například v souboru `testPrg.zig` lze pak provést na serveru `merlin` pomocí příkazu:

```
zig run testPrg.zig < test.in > test.out
```

Tím lze jednoduše zkontrolovat, co by měl provést zadaný zdrojový kód, resp. vygenerovaný cílový kód. Je ale potřeba si uvědomit, že jazyk Zig je nadmnožinou jazyka IFJ24, a tudíž může zpracovat i konstrukce, které nejsou v IFJ24 povolené (např. bohatší syntaxe a sémantika většiny příkazů, či dokonce zpětné nekompatibility). Výčet těchto odlišností bude uveden v Moodle IFJ a můžete jej diskutovat na fóru předmětu IFJ.

[18] Online dokumentace k Zig: `https://ziglang.org/documentation/0.13.0/`

10 Cílový jazyk IFJcode24

Cílový jazyk IFJcode24 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 IFJcode24 jsou podporovány jednořádkové komentáře začínající mřížkou (#). Kód v jazyce IFJcode24 začíná úvodním řádkem s tečkou následovanou jménem jazyka:

```
.IFJcode24
```

10.1 Hodnotící interpret `ic24int`

Pro hodnocení a testování mezikódu v IFJcode24 je k dispozici interpret pro příkazovou řádku (`ic24int`):

```
ic24int 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 IFJcode24.
- 52 - chyba při sémantických kontrolách vstupního kódu v IFJcode24.
- 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. IFJcode24 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 IFJcode24 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 IFJ24.

Zápis každé konstanty v IFJcode24 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 (zaokrouhlený k $-\infty$) přiřadí do proměnné ⟨*var*⟩ typu int. Dělení nulou způsobí chybu 57.

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

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

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

AND/OR/NOT ⟨*var*⟩ ⟨*symb*$_1$⟩ ⟨*symb*$_2$⟩ Základní booleovské operátory
Aplikuje konjunkci (logické A)/disjunkci (logické NEBO) na operandy typu bool ⟨*symb*$_1$⟩ a ⟨*symb*$_2$⟩ nebo negaci na ⟨*symb*$_1$⟩ (NOT má pouze 2 operandy) a výsledek typu bool zapíše do ⟨*var*⟩.

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

INT2FLOAT ⟨*var*⟩ ⟨*symb*⟩ Převod celočíselné hodnoty na desetinnou
Převede celočíselnou hodnotu ⟨*symb*⟩ na desetinné číslo a uloží je do ⟨*var*⟩.

FLOAT2INT ⟨*var*⟩ ⟨*symb*⟩ Převod desetinné hodnoty na celočíselnou (oseknutí)
Převede desetinnou hodnotu ⟨*symb*⟩ na celočíselnou oseknutím desetinné části a uloží ji do ⟨*var*⟩.

INT2CHAR ⟨*var*⟩ ⟨*symb*⟩ Převod celého čísla na znak
Číselná hodnota ⟨*symb*⟩ je dle ASCII převedena na znak, který tvoří jednoznakový řetězec přiřazený do ⟨*var*⟩. Je-li ⟨*symb*⟩ mimo interval ⟨0; 255⟩, dojde k chybě 58.

STRI2INT ⟨*var*⟩ ⟨*symb*$_1$⟩ ⟨*symb*$_2$⟩ Ordinální hodnota znaku
Do ⟨*var*⟩ uloží ordinální hodnotu znaku (dle ASCII) v řetězci ⟨*symb*$_1$⟩ na pozici ⟨*symb*$_2$⟩ (indexováno od nuly). Indexace mimo daný řetězec vede na chybu 58.

INT2FLOATS/FLOAT2INTS/ INT2CHARS/STRI2INTS Zásobníkové verze konverzních instrukcí

10.4.4 Vstupně-výstupní instrukce

READ ⟨*var*⟩ ⟨*type*⟩ Načtení hodnoty ze standardního vstupu
Načte jednu hodnotu dle zadaného typu ⟨*type*⟩∈ {int, float, string, bool} (včetně případné konverze vstupní hodnoty float při zadaném typu int) a uloží tuto hodnotu do proměnné ⟨*var*⟩. Formát hodnot je kompatibilní s chováním vestavěných funkcí **`ifj.readstr`**, **`ifj.readi32`** a **`ifj.readf64`** jazyka IFJ24.

WRITE ⟨*symb*⟩ Výpis hodnoty na standardní výstup
Vypíše hodnotu ⟨*symb*⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **`ifj.write`** jazyka IFJ24 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 IFJcode24. V případě skoku na neexistující návěští dojde k chybě 52.

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

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

JUMPIFEQ ⟨*label*⟩ ⟨*symb*$_1$⟩ ⟨*symb*$_2$⟩ Podmíněný skok na návěští při rovnosti
Pokud jsou ⟨*symb*$_1$⟩ a ⟨*symb*$_2$⟩ stejného typu nebo je některý operand nil (jinak chyba 53) a zároveň se jejich hodnoty rovnají, tak provede skok na návěští ⟨*label*⟩.

JUMPIFNEQ ⟨*label*⟩ ⟨*symb*$_1$⟩ ⟨*symb*$_2$⟩ Podmíněný skok na návěští při nerovnosti
Jsou-li ⟨*symb*$_1$⟩ a ⟨*symb*$_2$⟩ stejného typu nebo je některý operand nil (jinak chyba 53), ale různé hodnoty, tak provede skok na návěští ⟨*label*⟩.

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

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

10.4.8 Ladicí instrukce

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

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

11 Pokyny ke způsobu vypracování a odevzdání

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

11.1 Obecné informace

Za celý tým odevzdá projekt vedoucí. Všechny odevzdané soubory budou zkomprimovány programem ZIP, TAR+GZIP, nebo TAR+BZIP do jediného archivu, který se bude jmenovat `xlogin99.zip`, `xlogin99.tgz`, nebo `xlogin99.tbz`, kde místo zástupného řetězce `xlogin99` použijte školní přihlašovací jméno **vedoucího** týmu. Archiv nesmí obsahovat adresářovou strukturu ani speciální či spustitelné soubory. Názvy všech souborů budou obsahovat pouze písmena[19], číslice, tečku a podtržítko (ne mezery!).

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

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

11.2 Dělení bodů

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

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

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

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

12 Požadavky na řešení

Kromě požadavků na implementaci a dokumentaci obsahuje tato kapitola i několik rad pro zdárné řešení tohoto projektu a výčet rozšíření za prémiové body.

[19] 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 *login_vedoucího*, varianta *X*“ a výčet identifikátorů implementovaných rozšíření.
- Rozdělení práce mezi členy týmu (uveďte kdo a jak se podílel na jednotlivých částech projektu; povinně zdůvodněte odchylky od rovnoměrného rozdělení bodů).
- Diagram konečného automatu, který specifikuje lexikální analyzátor.
- LL-gramatiku, LL-tabulku a precedenční tabulku, podle kterých jste implementovali váš syntaktický analyzátor.
- Stručný popis členění implementačního řešení včetně názvů souborů, kde jsou jednotlivé části včetně povinných implementovaných metod překladače k nalezení.

Dokumentace nesmí:

- obsahovat kopii zadání či text, obrázky[20] 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 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 IFJcode24 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

[20] 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 *Moodle* IFJ bude odkazován skript `is_it_ok.sh` na kontrolu většiny formálních požadavků odevzdávaného archivu, který doporučujeme využít.

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

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

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

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

12.6 Pokusné odevzdání

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

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

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

12.7 Registrovaná rozšíření

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

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

12.7.1 Bodové hodnocení některých rozšíření jazyka IFJ24

Popis rozšíření vždy začíná jeho identifikátorem a počtem možných získaných bodů. Většina těchto rozšíření je založena na dalších vlastnostech jazyka Zig, nemusí však jazyku Zig odpovídat v celém jeho rozsahu. Podrobnější informace lze získat ze specifikace jazyka Zig[2]. Pokud váháte např. s určením vhodného chybového kódu, využijte diskuzní fórum (obvykle budeme akceptovat všechny možnosti, které „dávají smysl“). Do dokumentace je potřeba (kromě zkratky na úvodní stranu) také uvést, jak jsou implementovaná rozšíření řešena.

ORELSE (+0,5 bodu) Překladač bude podporovat binární operátor **`orelse`** (operátor je zleva asociativní a má prioritu 2,5, tj. nižší než **`+`** a vyšší než **`==`**). Nechť *a*, *b* jsou termy. Typ *a* zahrnuje **`null`** (má předponu **`?`**), typ *b* je shodný s typem *a*, ale nemusí zahrnovat **`null`**. (Pokud si typy neodpovídají, chyba 7.) Výraz *a* **`orelse`** *b* pak nabývá hodnoty *b* právě tehdy, když hodnota *a* je **`null`**, jinak nabývá hodnoty *a*. Pokud typ *b* zahrnuje **`null`** nebo je *b* literálem **`null`**, výsledný typ také zahrnuje **`null`** (je shodný s typem *a*), v opačném případě je výsledný typ bez **`null`**. Příklad:

```
const noVal : ?i32 = null;
const maybeVal : ?i32 = 123;
const x = noVal orelse 42;
// 'x' ma typ i32 a hodnotu 42
const y = noVal orelse null;
// 'y' ma typ ?i32 a hodnotu null
const z = noVal orelse maybeVal;
// 'z' ma typ ?i32 a hodnotu 123
```

UNREACHABLE (+0,5 bodu) Pro implementaci tohoto rozšíření je nejprve nutná podpora ORELSE. Překladač bude podporovat speciální literál **`unreachable`** a ve výrazech s proměnnými unární postfixový operátor '**`.?`**' (operátor má prioritu 0, tj. vyšší než **`*`**). Literál **`unreachable`** je možné použít na pravé straně operátoru **`orelse`** nehledě na typ levé strany. Výsledným typem je typ levé strany, který ale nezahrnuje **`null`**. Pokud se levá strana při běhu programu vyhodnotí na **`null`**, program vypíše řetězec `panic: reached unreachable code` a končí s návratovým kódem 57. Výraz `a.?` je sémanticky ekvivalentní s `(a orelse unreachable)`. Příklad:

```
const maybeVal : ?i32 = 42;
const noVal : ?i32 = null;
const x = maybeVal orelse unreachable;
const y = maybeVal.?;
// 'x' ma typ i32 a hodnotu 42, taktez 'y'
const z = noVal orelse unreachable;
// 'z' ma odvozen typ i32 a program je prelozen,
// ale po spusteni zde konci s chybou
```

BOOLTHEN (+1,0 bodu) Podpora typu **bool**, booleovských hodnot **true** a **false**, booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**!**, **and** a **or** včetně zkratového vyhodnocení), jejichž priorita a asociativita odpovídá jazyku Zig[2]. Pravdivostní hodnoty lze porovnávat jen operátory **==** a **!=**. Podporujte výpisy hodnot typu **bool** a přiřazování výsledku booleovského výrazu do proměnné. Podporujte zkrácený (bez **else**-části) a rozšířený podmíněný příkaz **if-else if-else**. Dále podporujte podmíněný (ternární) výraz

```
if (pravdivostni_vyraz) vyraz1 else vyraz2
```

FOR (+0,25 bodu) Překladač bude podporovat cyklus **for**, který pracuje nad řetězci:

for (*výraz***) |***id***| {**
sekvence_příkazů
}

kde *výraz* je řetězcový literál nebo výraz typu **[]u8**. V těle cyklu bude dostupná proměnná *id* typu **i32**, která bude v každé iteraci postupně nabývat ordinálních hodnot (ASCII) jednotlivých znaků řetězce.

Pro zajištění kompatibility s jazykem Zig bude překladač podporovat také speciální vestavěnou funkci:

@as(i32, *id***)**

která vrací hodnotu typu **i32**. Upozorňujeme, že bez použití této funkce (při přímém přístupu k *id*) nemusí být vaše programy přeložitelné překladačem Zig (v jazyce Zig má proměnná typ **u8**, který v IFJ24 není podporován)!

id může být i pseudoproměnná '_' (podtržítko), pokud se v sekvenci příkazů nepoužívá. Podporujte i příkazy **break** a **continue**. Příklad:

```
const str: []u8 = ifj.string("retezec");
for (str) |temp| {
    const character: i32 = @as(i32, temp);
    if (character == 32) {
        ifj.write("Nemam rad mezery.\n");
        continue;
    } else {}
    ifj.write("Ordinalni hodnota: ");
    ifj.write(character);
    ifj.write(".\n");
}
```

WHILE (+1,0 bodu) Překladač bude podporovat rozšířenou verzi příkazu cyklu **`while`** s možností použití návěští, „continue expression“, větve **`else`** a podmiňování neprázdnou hodnotou.

název`: while (`*výraz*`) : (`*příkaz* nebo `{`*sekvence_příkazů*$_{cnt}$`})`
`{`
  sekvence_příkazů$_1$
`}`
`else`
`{`
  sekvence_příkazů$_2$
`}`

kde *název* je **volitelné** návěští cyklu, které je možné použít pro lepší řízení toku s vnořenými cykly[21]. Podporujte příkazy **`break`** a **`continue`**, za nimi může volitelně následovat *název* cyklu, který se má ukončit (resp. ve kterém se má přejít k další iteraci).

Část ’`: (`*příkaz* nebo `{`*sekvence_příkazů*$_{cnt}$`})`’ (tzv. continue expression) je také **volitelná**. Pokud je přítomna, uvedený *příkaz* nebo *sekvence_příkazů*$_{cnt}$ se vykoná bezprostředně po konci každé iterace (tj. ještě před vyhodnocením podmínky pro další iteraci). Nevykoná se ale, pokud je cyklus ukončen příkazem **`break`**.

Cyklus může **volitelně** obsahovat také větev **`else`**. Ta se provede, jakmile je poprvé podmínka cyklu vyhodnocena na **`false`** (tj. před jeho ukončením, ale ne v případě, že je cyklus ukončen příkazem **`break`**).

Návěští, continue expressions i větev **`else`** je možné použít také u cyklů podmiňených neprázdnou hodnotou. Hlavička cyklu může tedy mít alternativně podobu:

název`: while (`*výraz_s_null*`) |`*id_bez_null*`| : (`*příkaz* nebo `{`*sekvence_příkazů*`})`

Pro chování cyklu pak obdobně platí požadavky uvedené výše. V části *příkaz* nebo *sekvence_příkazů*$_{cnt}$ je možné použít proměnnou *id_bez_null*.

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

13 Opravy zadání

- 25. 9. 2024 – Oprava příkladu s escape sekvencí v řetězcovém literálu (odstranění přebytečných složených závorek).

[21] Pokud implementujete také rozšíření FOR, bylo by vhodné návěští a řízení toku podporovat také zde, ale testováno to nebude.