ZADÁNÍ PROJEKTU Z PŘEDMĚTU VYPE

Zbyněk Křivka
email: krivka@fit.vutbr.cz
23. září 2015

1 Obecné informace

Název projektu:	Implementace překladače programovacího jazyka VYPe15.
Informace:	diskusní fórum VYPe v IS FIT, stránky projektu do VYPe
Stránky projektu:	`http://www.fit.vutbr.cz/study/courses/VYPe/public/project/`
Datum odevzdání:	pátek 1. ledna 2016, 23:59
Způsob odevzdání:	prostřednictvím IS FIT do datového skladu předmětu VYPe

Řešitelské týmy:

- Projekt budou řešit **dvoučlenné týmy**.
- Registrace se provádí přihlášením na příslušnou variantu termínu "Project (Compiler)" v IS FIT. Registrace je dvoufázová. V první fázi vytvoří dvojice studentů týmy (nová funkcionalita IS FIT). Jeden ze studentů je tzv. *kapitán* a má na starost složení týmu a přihlašování týmu na vybranou variantu projektu, což je druhá fáze. Případná vzájemná elektronická komunikace mezi vyučujícími a týmem bude zasílána v kopii všem členům týmu. Pro řešení obecných dotazů preferujte fórum předmětu. Na správném položení dotazu si dejte záležet.

Hodnocení:

- Každý člen týmu získá maximálně 20 bodů (17 funkčnost projektu, 3 dokumentace).
- Max. 25% bodů Vašeho individuálního základního hodnocení navíc za tvůrčí přístup (různá rozšíření apod.).
- Dokumentace bude hodnocena nejvýše třetinou bodů z hodnocení funkčnosti projektu a bude také reflektovat procentuální rozdělení bodů.

2 Zadání

Pro níže popsaný jednoduchý programovací jazyk VYPe15 definujte vhodnou gramatiku a vytvořte syntaxí řízený překladač (spustitelný soubor nebo skript jménem `vype`), který načte zdrojový soubor zapsaný v jazyce VYPe15 a přeloží jej do cílového jazyka

symbolických adres (MIPS) pro procesor MIPS32-Lissom[1] tak, aby byl přeložený program simulovatelný na dodaných nástrojích. Jestliže proběhne činnost vašeho překladače bez chyb, vrací se návratová hodnota `0` (nula). Jestliže došlo k nějaké chybě, vrací se návratová hodnota následovně:

- 1 = chyba v programu v rámci lexikální analýzy (chybná struktura aktuálního lexému).
- 2 = chyba v programu v rámci syntaktické analýzy (chybná syntaxe programu).
- 3 = chyba v programu v rámci sémantických kontrol (nedeklarovaná proměnná, nekompatibilita typů atd.).
- 4 = chyba při generování výstupních instrukcí jazyka symbolických adres (např. nedostatek staticky alokované paměti)
- 5 = interní chyba překladače, tj. neovlivněná vstupním programem (např. chyba alokace paměti nutné pro překlad, chyba při otvírání vstupního či výstupního souboru atd.).

Jméno souboru se vstupním programem v jazyce VYPe15 bude předáno jako první parametr na příkazové řádce. Druhý (nepovinný) parametr určuje jméno výstupního souboru s přeloženým programem v jazyce MIPS (implicitně `out.asm`). Oba parametry je možné zadat relativní i absolutní cestou. Existující soubor pro výstup bude bez varování přepsán. Všechna chybová hlášení vašeho překladače budou prováděna na standardní chybový výstup; tj. bude se jednat o konzolovou aplikaci, nikoliv o aplikaci s grafickým uživatelským rozhraním.

Veškeré regulární výrazy použité v dalším textu jsou vždy zapsány podle pravidel GNU pro rozšířené regulární výrazy. Případné mezery v regulárních výrazech jsou většinou (kromě případů, kdy jsou nutné pro oddělení dvou po sobě jdoucích slov) sázeny pouze pro zlepšení čitelnosti a nejsou součástí regulárních výrazů. Klíčová slova jsou sázena tučně. Identifikátory a neterminální symboly jsou psány kurzívou. Slovo *id* vždy zastupuje libovolný identifikátor. Některé lexémy jsou pro zvýšení čitelnosti v uvozovkách, přičemž znak uvozovek není v takovém případě součástí jazyka! Bližší popis regulárních výrazů lze nalézt například po zadání příkazu `info grep` na linuxovém systému.

3 Popis programovacího jazyka

Jazyk VYPe15 je inspirován podmnožinou jazyka C.

3.1 Obecné vlastnosti a datové typy

Programovací jazyk VYPe15 je case-sensitive (rozlišují se velká a malá písmena jak u klíčových slov, tak i u identifikátorů) a je jazykem typovaným, takže každá proměnná má předem určen datový typ svou deklarací.

[1] Implementace MIPS32 výzkumnou skupinou Lissom. Nástroje jako assembler, linker a instrukční simulátor budou k dispozici na stránkách projektu do VYPe.

- *Identifikátor* je definován jako neprázdná posloupnost číslic, písmen (malých i velkých) a znaku podtržítka ("_") začínající písmenem nebo podtržítkem. Jazyk VYPe15 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam, a proto se nesmějí vyskytovat jako identifikátory: **`char`**, **`else`**, **`if`**, **`int`**, **`return`**, **`string`**, **`void`** a **`while`**. Dále jazyk obsahuje několik rezervovaných slov: **`break`**, **`continue`**, **`for`**, **`short`** a **`unsigned`**. Dále není možné definovat více proměnných stejného jména v rámci stejného jmenného prostoru.
- *Datové typy* pro jednotlivé uvedené literály jsou označeny **`int`**, **`char`** a **`string`**. Typy se používají ve spojitosti s proměnnými, s typy výsledků funkcí a s jejich parametry (dále označovány jako *datový_typ*). Dále existuje datový typ **`void`** (nepatří do *datový_typ*), který označuje typ prázdný (funkce nevrací žádná data). *datový_typ* nebo **`void`** označujeme jako *typ*.
- *Celočíselný literál* (konstanta v desítkové soustavě) je definován výrazem **`[0-9]+`** a je reprezentován znaménkovým typem **`int`** s rozsahem 32 bitů.
- *Znaková konstanta* obsahuje jeden *tisknutelný znak* ohraničený apostrofy (`'`, ASCII hodnota 39). *Tisknutelný znak* může být tvořen znakem s ASCII hodnotou větší jak 31 (kromě 34 a 39) nebo jednou escape sekvencí. Možné escape sekvence jsou: **`\n`**, **`\t`**, **`\\`**, **`\"`**, **`\'`**. Jejich význam se shoduje s odpovídajícími znakovými konstantami jazyka C.
- *Řetězcová konstanta* je tvořena *tisknutelnými znaky* (včetně escape sekvencí) a je ohraničena uvozovkami (`"`, ASCII hodnota 34) z obou stran. Jedná se o tzv. null-terminated řetězec, který je interně ukončen bajtem s ASCII hodnotou 0. Délka této konstanty není omezena (snad jen velikostí volné paměti programu; dostatek volné paměti není nutno kontrolovat).
- Jazyk VYPe15 podporuje *řádkové komentáře*. Komentář je označen pomocí dvojice lomítek ("**`//`**") a za komentář je považováno vše, co následuje až do konce řádku.
- Jazyk VYPe15 podporuje i *blokové komentáře*. Tyto komentáře začínají dvojicí znaků "**`/*`**" a jsou ukončeny dvojicí znaků "**`*/`**". Vnořené blokové komentáře neuvažujte.

4 Struktura jazyka

VYPe15 je strukturovaný programovací jazyk podporující definice proměnných a deklarace a definice funkcí včetně jejich rekurzivního volání. Vstupním bodem řídicího programu je hlavní funkce programu jménem **`main`**.

4.1 Základní struktura jazyka

Program se skládá z globálních deklarací funkcí a definic funkcí (nelze je zanořovat). Některé příkazy jazyka VYPe15 jsou ukončeny středníkem ("`;`", tzv. terminátor jednoduchých příkazů). Kromě komentářů také ignorujte všechny bílé znaky (mezery, tabulátory a konce řádku), které se mohou vyskytovat v libovolném množství mezi všemi lexémy i na začátku a konci zdrojového textu.

4.2 Deklarace a definice funkcí

Každá funkce musí být definována právě jednou, deklarována maximálně jednou (deklarace jsou nepovinné). Deklarace funkcí slouží především pro křížové rekurzivní volání dvou či více funkcí navzájem. Definice nebo alespoň deklarace funkce musí být vždy k dispozici před prvním voláním funkce (volání funkce je definováno v sekci 4.4). Deklarace a definice funkce má následující tvar:

- *Deklarace funkce* je ve tvaru:
  typ id **\(** *seznam_typů_parametrů* **\)** **;**
- *Definice funkce* je konstrukce ve tvaru:
  typ id **\(** *seznam_parametrů* **\)** **{**
  (*stmt* **)***
  }
  - Deklarace jednotlivých formálních parametrů jsou odděleny čárkou, za poslední z nich se čárka neuvádí, takže *seznam_typů_parametrů* obsahuje pouze čárkou oddělené typy. *seznam_parametrů* má tvar:
    (*datový_typ id* **(** **,** *datový_typ id* **)*** **)**
    Parametry jsou vždy předávány hodnotou. Místo prázdného seznamu typů parametrů či prázdného seznamu parametrů se použije typ **`void`**.
  - Tělo funkce, popsané regulárním výrazem **(***stmt***)*** , je sekvence příkazů (může být i prázdná). Popis jejich syntaxe a sémantiky je v sekci 4.4.

Z hlediska sémantiky je potřeba kontrolovat, zda souhlasí počet parametrů a pořadí typů v deklaraci funkce a v hlavičce definice funkce. Případná deklarace funkce musí předcházet její definici. Každá deklarovaná funkce musí být definována. Přetěžování funkcí není povoleno (až na vestavěnou funkci **`print`** a bonusové rozšíření OVERLOAD).

4.3 Hlavní funkce programu

Každý program musí obsahovat právě jednu hlavní funkci pojmenovánou **`main`** se signaturou **`int main \( void \)`**, jinak se jedná o sémantickou chybu. Struktura definic proměnných a příkazů je uvedena v následujících sekcích.

4.4 Syntaxe a sémantika příkazů

Příkazem *stmt* se rozumí:

- *Definice lokálních proměnných*:
  datový_typ id **(** **,** *id* **)*** **;**
  Definované lokální proměnné jsou platné od tohoto příkazu až do konce bloku[2], ve kterém byly definovány. Jazyk VYPe15 umožňuje překrývání proměnných v různých úrovních zanoření (přes stejné jméno). Takováto stejně pojmenovaná proměnná z nadbloku je dočasně neviditelná. Číselné proměnné jsou implicitně inicializovány

[2] *Blok* je sekvence příkazů ohraničená složenými závorkami tvaru: **{** **(** *stmt* **)*** **}** a tvoří tzv. *jmenný prostor*.

na hodnotu 0, znakové na `'\0'` a řetězcové na `""`. Definice proměnné stejného jména jako již deklarovaná funkce či proměnná definovaná v aktuálním bloku není povolena.

- *Příkaz přiřazení*:
  id **=** *výraz* **;**
  Sémantika příkazu je následující: Příkaz provádí přiřazení hodnoty pravého operandu do levého operandu. Platí pro něj: levý operand musí být deklarován a také musí být stejného typu jako pravý operand, a to **int**, **char**, nebo **string**. Levý operand musí být vždy pouze proměnná (tzv. l-hodnota). Pokud bude l-hodnotou proměnná typu **string**, tak bude překladem zajištěna případná realokace paměti potřebné pro uložení celého pravého operandu, protože se provádí paměťová kopie řetězce (nikoli pouze odkazu/reference).
- *Podmíněný příkaz*:
  if \(*výraz* **\)**
  { ($stmt_1$ **)* }**
  else
  { ($stmt_2$ **)* }**
  Sémantika příkazu je následující: Nejprve vyhodnotí daný *výraz*. Pokud výsledek výrazu je jiného typu než **int**, jedná se o sémantickou chybu. Jinak, pokud výsledná hodnota výrazu je rovna číslu `0` (nula), výraz je považován za nepravdivý. Za pravdivý je výraz považován pro všechny nenulové hodnoty jeho výsledku.
  Pokud je hodnota výrazu pravdivá, vykoná se první sekvence příkazů **(** $stmt_1$ **)***, jinak se vykoná sekvence příkazů **(** $stmt_2$ **)***.
- *Příkaz cyklu* **while**:
  while \(*výraz* **\)**
  { (*stmt* **)* }**
  Sémantika příkazu cyklu **while** je následující: Pravidla pro určení pravdivosti výrazu jsou stejná jako v případě podmíněného příkazu. Příkaz cyklu **while** opakuje provádění sekvence příkazů **(** *stmt* **)*** tak dlouho, dokud je hodnota výrazu *výraz* pravdivá.
- *Volání vestavěné a uživatelem definované funkce*:
  *id_funkce***\((** *výraz* **(,** *výraz* **)*)? \) ;**
  Sémantika volání funkce *id_funkce* je následující: Příkaz je vlastně vyhodnocením výrazu volání funkce se zahozením výsledné hodnoty, pokud nebyla funkce typu **void**. Nejprve jsou vyhodnoceny postupně všechny výrazy pro parametry a jejich hodnoty jsou předány hodnotou do funkce, kde jsou přístupné přes formální jména parametrů funkce (viz definice funkce). Typy a počet skutečných parametrů musí odpovídat dané deklaraci (potažmo definici) funkce. V případě funkce bez parametrů se uvedou za jménem funkce pouze prázdné závorky. Pokud je některý parametr jiného typu, než který je očekáván, nastane sémantická chyba. Pak je předáno řízení do těla funkce. Po návratu z těla funkce (viz příkaz **return**) dojde k pokračování běhu programu bezprostředně za příkazem volání funkce. Pokud není funkce součástí výrazu, tak je její návratová hodnota zahozena, aniž by se generovalo chybové hlášení. Funkce lze volat rekurzivně do libovolné úrovně zanoření (omezeno pouze množ-

Pr.	Operátory	Asoc.	Význam	Typy
9	`()`	—	přetypování/volání funkce	
8	`!`	unární	logická negace (prefixová)	$N \to N$
7	`*` `/` `%`	$\to$	celočíselné multiplikativní	$N \times N \to N$
6	`+` `-`	$\to$	celočíselné aditivní	$N \times N \to N$
5	`<` `<=` `>` `>=`	$\to$	relační	$T \times T \to \{0,1\}$
4	`==` `!=`	$\to$	porovnání	$T \times T \to \{0,1\}$
1	`&&`	$\to$	logické AND	$N \times N \to \{0,1\}$
0	`||`	$\to$	logické OR	$N \times N \to \{0,1\}$

Tabulka 1: Operátory jazyka VYPe15

stvím volné paměti). Sémantika vestavěných funkcí bude popsána v kapitole 4.6.

- *Příkaz návratu z funkce*:
  `return (` *výraz* **`)? ;`**
  Příkaz slouží pro ukončení funkce (tzv. volané funkce, angl. *callee*) a vrácení funkční hodnoty do volající funkce (angl. *caller*). Jeho sémantika je následující: Dojde k vyhodnocení výrazu *výraz* (je-li přítomen) pro získání návratové hodnoty, okamžitému ukončení provádění těla funkce, návratu návratové hodnoty (není-li funkce typu **`void`**) a návratu do místa volání. V ostatních případech nejednotnosti typů se jedná o sémantickou chybu. Funkce bez návratové hodnoty (typu **`void`**) vyžaduje ukončení příkazem **`return`** bez parametru *výraz*. V těle funkce **`main`** provede příkaz "**`return`** *výraz* **`;`**" vyhodnocení návratové hodnoty a ukončení programu (viz instrukce **`BREAK`**). Pokud funkce (platí analogicky i pro funkci **`main`**) doběhne na svůj konec bez vykonání příkazu **`return`**, tak podle svého návratového typu vrací implicitní hodnotu 0, `'\0'` nebo `""`.

4.5 Výrazy

Výrazy jsou tvořeny číselnými, znakovými a řetězcovými konstantami, proměnnými, závorkami, voláním funkcí, přetypováním nebo výrazy tvořenými aritmetickými, logickými a relačními operátory. Binární aritmetické operátory i binární logické spojky jsou zleva asociativní.

Seznam operátorů včetně priorit (od nejvyšší k nejnižší) je uveden v tab. 1, kde ve sloupci **Typy** jsou použity následující zkratky: N = **`int`**, T = *datový_typ*. Detailní popis těchto operátorů následuje ve zbytku této sekce.

4.5.1 Aritmetické, logické a relační operátory

Pro operátory **`+`**, **`-`**, **`*`** platí: Pokud jsou oba operandy typu **`int`**, výsledek je typu **`int`**.

Operátor **`/`** značí celočíselné dělení dvou číselných operandů (výsledek je typu **`int`**). Při dělení nulou se neprovede nic (viz instrukce **`DIV`**).

U operátoru **`%`** se jedná o získání celočíselného zbytku po operaci celočíselného dělení (viz instrukce **`DIV`**). Výsledek je typu **`int`**, jsou-li oba operandy také typu **`int`**.

Pro operátory `<`, `>`, `<=`, `>=`, `==`, `!=` platí: Pokud je první operand stejného typu jako druhý operand, a to **`char`**, **`int`** nebo **`string`**, výsledek je typu **`int`**. U řetězců se porovnání provádí lexikograficky. Výsledkem porovnání je celočíselná hodnota `1` (jedna) v případě pravdivosti relace, jinak `0`. Operátory mají stejnou sémantiku jako v jazyce C.

Logické operátory **`!`**, **`&&`** a **`||`** přijímají číselné operandy typu **`int`** (včetně výsledků jiných logických či relačních operací). Nulová hodnota má význam nepravdivostní hodnoty, jakákoli jiná hodnota je považována za pravdivou. Výsledek aplikace logického operátoru je buď pravdivý (hodnota **`1`**), nebo nepravdivý (hodnota **`0`**).

Jiné, než uvedené kombinace typů ve výrazech pro dané operátory, jsou považovány za chybu.

4.5.2 Závorky

Operátor explicitního přetypování se zapisuje jako **`\(`***datový_typ***`\)`***výraz* a umožňuje přetypovat **`char`** na **`string`** délky 1 a **`char`** na **`int`** (8-bitová ASCII hodnota znaku (bezznaménková) rozšířená na 32 bitů). Naopak **`int`** lze explicitně přetypovat na **`char`** výběrem nejméně významného bajtu.

Výsledkem výrazu volání funkce je její návratová hodnota.

Prioritu operátorů lze explicitně upravit závorkováním podvýrazů. Tabulka 1 udává priority operátorů (nahoře nejvyšší).

4.6 Vestavěné funkce

Jazyk VYPe15 bude poskytovat tyto signaturami zadané základní vestavěné funkce (tj. funkce, které jsou přímo implementovány v překladači a jsou využitelné v programovacím jazyce VYPe15 bez jejich definice či deklarace):

- **`void print\(`** *datový_typ* **`( , `** *datový_typ* **`)* \)`** – Funkce tiskne postupně všechny své parametry na výstup pomocí speciálních instrukcí pro výstup. Parametrů je libovolný počet (minimálně jeden) a parametry jsou libovolného datového typu.
- **`char read_char\( void \)`** – Načte vstup od uživatele ukončený koncem řádku (**`EOL`**[3]) pomocí speciální instrukce pro vstup znaku.
- **`int read_int\( void \)`** – Načte vstup od uživatele ukončený koncem řádku (**`EOL`**) pomocí speciální instrukce pro vstup celého čísla rozsahu **`int`** (bez kontroly přetečení).
- **`string read_string\( void \)`** – Načte vstup od uživatele ukončený koncem řádku (**`EOL`**) pomocí speciální instrukce pro vstup řetězce (bez **`'\n'`**).
- **`char get_at\(string , int \)`** – Funkce vrací znak v řetězci zadaném prvním parametrem na indexu zadaném druhým parametrem (indexováno od nuly). Neprovádí se žádná kontrola mezí, takže v případě záporné nebo příliš vysoké hodnoty indexu je chování nedefinováno.

[3] Simulátor instrukční sady MIPS32-Lissom (podle OS, na kterém je spuštěný) považuje za **`EOL`** znak $\langle LF \rangle$ na unixových systémech a znaky $\langle CR \rangle\langle LF \rangle$ na Windows.

- **`string set_at\(string , int , char \)`** – Funkce vrací řetězec, jenž vznikl modifikací řetězce z prvního argumentu přepsáním znaku na indexu zadaném druhým parametrem (indexováno od nuly) znakem, který je zadán posledním parametrem. Opět není prováděna žádná kontrola mezí.
- **`string strcat\(string , string \)`** – Funkce vrací řetězec, jenž je konkatenací řetězce z prvního argumentu a řetězce z druhého argumentu.

4.7 Poznámky k implementaci

Návrh a realizaci kompilátoru můžete provést libovolným způsobem. Kvůli rozsáhlosti kompilátoru a současným trendům v oblasti překladačů doporučujeme maximální využití automatizovaných nástrojů pro podporu tvorby kompilátorů. Především se jedná o využití nástrojů **`flex`** (novější implementace nástroje **`lex`**) pro tvorbu lexikálního analyzátoru a **`bison`** (novější **`yacc`**) pro vytváření syntaktického a sémantického analyzátoru využívajícího metodu zdola nahoru (LALR). Vzhledem k uvolnění podmínek na implementační jazyk lze použít i novější nástroje jako ANTLR (Java knihovny verze 3.5.2 a 4.5.1 jsou na serveru Merlin na cestě `/pub/courses/vyp/`). Ohledně použití jiných nástrojů tohoto druhu kontaktujte zadavatele projektu.

Podpora pro vstupně-výstupní vestavěné funkce je již integrována v cílovém jazyce MIPS (a dodaných nástrojích) pomocí speciálních instrukcí. Zbývá pouze zařídit správné provázání na úrovni generovaného kódu.

5 Příklady

Tato kapitola popisuje tři jednoduché příklady programů v jazyce VYPe15.

5.1 Výpočet faktoriálu (iterativně)

```
/* Program 1: Vypocet faktorialu (iterativne) */

int main(void) { // Hlavni telo programu
  int a, vysl;

  print("Zadejte cislo pro vypocet faktorialu");
  a = read_int();
  if (a < 0) {
     print("\nFaktorial nelze spocitat\n"); }
  else
  {
     vysl = 1;
     while (a > 0)
     {
        vysl = vysl * a;
        a = a - 1;
     } // endwhile
     print("\nVysledek je: ", vysl, "\n");
  } // endif
} // main
```

5.2 Výpočet faktoriálu (rekurzivně)

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

int factorial (int);
int factorial (int n)
{
  int decremented_n, temp_result;
  if (n < 2)
  {
     return 1;
  }
  else
  {
     decremented_n = n - 1;
     temp_result = factorial(decremented_n);
  }
  return (int)n * temp_result;
} // end of factorial

int main(void)
{
  int a; int vysl;
  print("Zadejte␣cislo␣pro␣vypocet␣faktorialu");
  a = read_int();
  if (a < 0)
  {
     print("\nFaktorial␣nelze␣spocitat\n");
  }
  else
  {
     print("\nVysledek␣je:", factorial(a), "\n");
  }
}
```

5.3 Práce s řetězci a vestavěnými funkcemi

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

int main(void) { //Hlavni telo programu
  string str1, str2;
  int p; char zero;

  str1 = "Toto␣je␣nejaky␣text";
  str2 = strcat(str1, ",␣ktery␣jeste␣trochu␣obohatime");
  print(str1, '\n', str2, "\n");

  str1 = read_string();
  while ((int)(get_at(str1, p)) != 0)
  {
    p = p + 1;
  }
  print("\nDelka␣retezce␣\"", str1, "\",␣je␣", p, "␣znaku.\n");
}
```

6 Cílový jazyk symbolických adres procesoru MIPS32

Nyní si stručně popišme jazyk symbolických adres pro Lissom implementaci procesoru MIPS32 (Release 1), zkráceně jazyk MIPS. Kapitola se nesnaží o vyčerpávající popis, jelikož existují podrobné manuály a dokumentace, kde lze dohledat detailní informace (viz sekce odkazů na stránce projektu do VYPe), ale o přehledovou a počáteční informaci pro rychlý úvod do problematiky.

Jako každý jazyk symbolických adres obsahuje i jazyk MIPS různé instrukce, identifikátory registrů, konstanty, direktivy a návěští. U instrukcí a identifikátorů registrů nezáleží na velikosti písmen (tzv. *case insensitive*), ale pozor na návěští, u kterých záleží na velikosti písmen (tzv. *case sensitive*). Každá instrukce musí být právě na jednom vlastním řádku, stejně jako direktivy či návěští. Jednořádkové i blokové komentáře lze psát stejným způsobem jako v jazyce C.

6.1 Základní charakteristika procesoru MIPS32-Lissom

Cílový procesor MIPS32-Lissom obsahuje[4]

- 32 univerzálních registrů o šířce 32 bitů (**`$0`** až **`$31`**)
- architekturní registry `rhi` a `rlo` pro operace dělení a násobení (z jazyka MIPS nedostupné)
- čítač programu (registr `PC`) (z jazyka MIPS opět nedostupný)
- paměť 1MB je organizovaná po bajtech (1 bajt = 8 bitů), nejmenší adresovatelná jednotka je 1 bajt, větší struktury jsou v paměti uloženy stylem *big endian*
- paměť je sdílená pro program i data (Von Neumannova architektura)
- datový zásobník chybí a je třeba jej simulovat
- chybí operace v plovoucí řádové čárce

6.2 Formát cílového souboru

Soubor pro překlad jazyka MIPS do simulatelného programu (pomocí nástroje assembler a linker) je rozdělen do několika sekcí. Každá sekce je uvozena speciální direktivou.

6.2.1 Direktivy

Direktivy jazyka MIPS umožňují zdrojový program rozdělit do různých sekcí (**`.text`** pro kód a **`.data`** pro data). Direktivou **`.org`** *číslo* lze určit absolutní pozici následujícího záznamu v paměti (maximálně jedna takováto direktiva v každé sekci). Tyto i další direktivy jako **`.ascii`**, **`.asciz`**, **`.byte`** a **`.int`** jsou popsány na `http://sourceware.org/binutils/docs-2.20/as/Pseudo-Ops.html`. Seznam povolených direktiv pro jazyk MIPS je dokumentován v Lissom Assembler Directives Reference (viz stránky projektu).

Příklad struktury souboru v jazyce MIPS:

[4] Detailní informace o modelu tohoto procesoru lze získat ze zveřejněného popisu v jazyce ISAC.

```
.text
.org 0
  MOVI $SP, 0x4000
jaL main
BREAK
main:
// ... save $RA ...
/* ... compute main body
   ... compute return value into $2 ... */
// ... load $RA ...
jr $RA
.ascii "ahoj\000"
.asciz "svete"
.int 1
.int 42
```

6.3 Stručný popis instrukční sady

Detailní popis instrukcí i jejich sémantiky lze najít na stránkách projektu do VYPe v manuálu "MIPS Technologies, Inc.: MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set, 2003 (Rev. 2.0)". Zaměřte se pouze na instrukce přítomné v procesoru verze 1 (viz stránky 24 až 27, tabulky 3-1 až 3-5, 3-7, 3-8 a instrukce **`BREAK`** z tabulky 3-9). Jejich konkrétní implementaci v MIPS32-Lissom lze ověřit v modelu procesoru v jazyce ISAC (opět viz stránky projektu).

Z instrukcí nejsou implementovány některé pro nás nepodstatné části instrukční sady jako instrukce pro operace nad desetinnými čísly, přerušení či výjimky (angl. *traps*), rozšíření pro koprocesor a systémová volání (například pro alokaci paměti, jejíž obsluhu je tedy třeba zajistit programově). Kvůli chybějícím výjimkám není možné na simulátoru MIPS32-Lissom detekovat chyby v době běhu programu a většina těchto chyb je bez varování přeskočena (např. pokus o dělení nulou tuto operaci přeskočí).

6.3.1 Speciální instrukce pro vstupně-výstupní operace

Následuje popis sémantiky a syntaxe speciálních instrukcí, kde *r*, *rd* a *rl* zastupují univerzální registr.

- *stdin* $\rightarrow$ *r* načte znak a uloží jej do registru *r*.
  `READ_CHAR` $*r*
- *stdin* $\rightarrow$ *r* načte hodnotu čísla a uloží ji do registru *r*.
  `READ_INT` $*r*
- *stdin* $\rightarrow$ *rd*, *rl* načte řetězec a uloží ho na adresu v paměti danou obsahem registru *rd* a do registru *rl* uloží délku načteného řetězce bez ukončujícího znaku \0.
  `READ_STRING` $*rd* **`,`** $*rl*
- *r* $\rightarrow$ *stdout* vypíše hodnotu registru *r* jako znak.
  `PRINT_CHAR` $*r*

Registr	Alias	Význam či rezervace
$0		obsahuje vždy nulu (zero register)
$1	$at	rezervováno pro jazyk MIPS
$28	$gp	globalní ukazatel (v případě globálních dat) (global pointer)
$29	$sp	ukazatel na vrchol zásobníku (stack pointer)
$30	$fp	ukazatel na kontext (frame pointer)
$31	$ra	nejaktuálnější návratová adresa (return address)

Tabulka 2: Vyhrazené registry instrukční sady procesoru MIPS32

- $r \rightarrow stdout$ vypíše hodnotu registru *r* jako číslo.
  `PRINT_INT` $*r*
- $r \rightarrow stdout$ vypíše řetězec na adrese dané obsahem registru *r*.
  `PRINT_STRING` $*r*

Každá načítaná hodnota (**`char`**/**`int`**/**`string`**) je na samostatném řádku a ukončena pomocí **`EOL`**. Konverze čísel je založena na funkci **`atoi`** jazyka C. Načítá se nejdelší platná sekvence znaků a zbytek znaků se až do konce řádku ignoruje. Pokud vstup neobsahuje na začátku žádný platný znak (je celý chybný), tak má výsledek hodnotu nula (**`0`**) či prázdného řetězce. Přetečení není řešeno (více viz model v jazyce ISAC).

6.3.2 Konec programu

Vyhodnocenou návratovou hodnotu funkce **`main`** uložte do registru $2 a ukončete program bezparametrickou instrukcí **`BREAK`**. Simulátor v textovém režimu nakonec vypíše obsahy všech registrů a pozici programového čítače.

6.3.3 Typická použití registrů pro MIPS ABI

Univerzální registry lze sice používat libovolným způsobem, ale pro lepší orientaci existuje jistý standard vyjádřený pojmenováním některých registrů. V tabulce 2 lze vidět nejdůležitější označení registrů, které doporučujeme ve vašem generátoru kódu reflektovat. Na stránce projektu je potom pro zájemce odkaz na detailnější popis MIPS ABI.

6.4 Dodané nástroje

Pro simulaci vašeho vygenerovaného kódu v jazyce MIPS je třeba nejprve provést jeho sestavení (assembler) a spojení (linker). Popišme si základní obsluhu nástrojů příkazové řádky.

6.4.1 Assembler a Linker

Předpokládejme, že váš výstupní program v jazyce MIPS je v souboru `out.asm`. Sestavení do objektového souboru `out.obj` se provede příkazem:

```
assembler2 -i out.asm -o out.obj
```

Poté je třeba vytvořit simulovatelný soubor pomocí příkazu:

```
linker out.obj -o out.xexe
```

6.4.2 Simulátor

Jelikož studenti nemají k dispozici fyzický procesor MIPS32, tak je výsledný kód pouze simulován dodaným instrukčním simulátorem. Simulátor začíná simulaci na paměťové adrese `0`, takže v případě, že váš program začíná na jiné adrese, tak je nutné vložit patřičnou skokovou instrukci. Samotnou simulaci z příkazové řádky se vstupem uloženým v souboru `input.in` spustíme příkazem:

```
intersim2 -i out.xexe -x report.xml -n mips < input.in
```

Na konci simulace bude vypsána tabulka s hodnotami jednotlivých univerzálních registrů, programového čitače a některé další statistiky.

7 Instrukce 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 ohodnotit!

7.1 Obecné informace

Za celý tým odevzdá projekt vedoucí. Všechny odevzdané soubory budou zkomprimovány programem TAR+GZIP či ZIP do jediného archivu, který se bude jmenovat `přihlašovací_jméno.tgz`, či `přihlašovací_jméno.zip`. Všechny názvy souborů uvnitř archivu budou odpovídat regulárnímu výrazu **`[A-Za-z_.0-9]+`**.

Celý projekt je třeba odevzdat v daném termínu (viz výše). Pokud tomu tak nebude, je projekt považován za neodevzdaný, nebo minimálně lze očekávat bodovou ztrátu úměrnou délce zpoždění. Stejně tak, pokud se bude jednat o plagiátorství jakéhokoliv druhu, je projekt hodnocený nula body a bude zváženo zahájení disciplinárního řízení.

7.2 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). Obsah souboru bude vypadat například takto:

```
xnovak01:60<LF>
xnovak02:40<LF>
```

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 členy týmu (i ty hodnocené 0%).

8 Požadavky na řešení

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

8.1 Závazné metody pro implementaci překladače

Při tvorbě lexikální a syntaktické analýzy doporučujeme využít některý z generátorů překladačů (Flex, Bison, ANTLR, ...). Je zakázáno použít nějakou existující přední či zadní část překladače (angl. *front-end* či *back-end*) jako například LLVM, GCC, SDCC a jiné. Implementace bude provedena **v libovolném jazyce** (např. C, C++, Java, Python[5]), v němž bude možno váš kompilátor přeložit na serveru `Merlin` (bez dodatečných nastavení), který je pro tento projekt brán jako referenční. Návrh implementace kompilátoru je zcela v režii řešitelských týmů. Součástí řešení bude soubor `Makefile`, ve kterém lze nastavit případné parametry překladu (viz `info gmake`). Pokud soubor pro sestavení cílového programu nebude obsažen nebo se na jeho základě nepodaří sestavit cílový program, nebude projekt hodnocený! Překlad nebo příprava spustitelného kompilátoru se provede příkazem `make`. Následuje možnost spuštění binárním programem nebo připraveným skriptem `vype`, který bude mít jeden povinný a jeden nepovinný parametr, jak je popsáno v kapitole 2.

8.2 Textová část řešení

Součástí řešení bude dokumentace, vypracovaná ve formátu PDF a uložená v jediném souboru **`dokumentace.pdf`**. Jakékoliv jiné formáty dokumentace, než předepsané, budou ignorovány, což povede ke ztrátě bodů za dokumentaci. Dokumentace bude vypracována v anglickém jazyce v rozsahu cca. 4-7 stran A4. **Dokumentace musí** povinně obsahovat:

- 1. strana: jména, příjmení a přihlašovací jména řešitelů + údaje o rozdělení bodů a výčet identifikátorů zvolených rozšíření.
- Popis přední části překladače včetně vypracované gramatiky a použitých metod a nástrojů.
- Popis vašeho způsobu řešení zadní části překladače - návrh, implementace, mezikód, tabulka symbolů, speciální použité techniky, optimalizace, algoritmy.
- Rozdělení práce mezi členy týmu (uveďte kdo a jakým způsobem se podílel na jednotlivých částech projektu).
- V případě nestandardní implementace uveďte popis spuštění na serveru Merlin či jiném prostředí, které bylo explicitně povoleno zadavatelem projektu.
- Literatura včetně citací u převzatých zdrojů (obrázků, statistik atd.).

Dokumentace nesmí:

- Obsahovat kopii zadání či text, obrázky[6] 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.).

[5] Jiný implementační jazyk raději konzultujte se zadavatelem projektu.
[6] Vyjma obyčejného loga fakulty na úvodní straně.

V rámci dokumentace bude rovněž vzat v úvahu stav kódu jako jeho čitelnost, srozumitelnost a dostatečné, ale nikoli přehnané komentáře.

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

Veškerá chybová hlášení vzniklá v průběhu činnosti překladače budou vždy vypisována na standardní chybový výstup. Přeložený řídicí program nebude během svého běhu na žádný výstup vypisovat žádné znaky či dokonce celé texty, které nejsou přímo předepsány řídicím programem v jazyce VYPe15. Základní testování bude probíhat pomocí automatu, který bude postupně spouštět sadu testovacích příkladů (kompilace příkladu vaším překladačem, sestavení a simulace dodanými nástroji) a porovnávat vaše výstupy s výstupy očekávanými. Pro porovnání výstupů bude použit program `diff` (viz `info diff`). Proto jediný neočekávaný znak, který dodaný simulátor podle Vámi přeloženého programu vytiskne, povede k nevyhovujícímu hodnocení aktuálního výstupu a tím snížení bodového hodnocení celého projektu.

8.3 Jak postupovat při řešení projektu

Teoretické znalosti, potřebné pro vytvoření projektu, získáte v průběhu semestru na přednáškách a diskuzním fóru VYPe. Je důležité, aby na řešení projektu spolupracoval celý tým. 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 některým členem týmu, lze řešit prostřednictvím souboru `rozdeleni` či osobní konzultací se zadavatelem projektu. Je ale nutné, abyste se vzájemně, nejlépe na pravidelných schůzkách týmu, ujišťovali o skutečném pokroku na jednotlivých částech projektu a případně včas přerozdělili práci. **Maximální počet bodů** získatelný na jednu osobu za celý projekt (včetně případných bonusových bodů) je **25**.

Nenechávejte řešení projektu až na poslední týden. Projekt je tvořen z několika částí (např. lexikální analýza, syntaktická analýza, sémantická analýza, intermediální reprezentace, tabulka symbolů, generování kódu, dokumentace, testování!) a dimenzován tak, aby některé části bylo možno navrhnout a implementovat již v průběhu semestru na základě dřívějších znalostí a znalostí získaných na přednáškách předmětu VYPe, na diskuzním fóru VYPe, na stránkách projektu a samostudiem.

8.4 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 unixovým koncem řádku, tj. znakem ⟨LF⟩).

V průběhu řešení bude postupně aktualizován ceník rozšíření a identifikátory rozšíření projektu (viz fórum k předmětu VYPe). V něm budou uvedena hodnocená rozšíření projektu, za která lze získat prémiové body. Na fórum předmětu můžete během semestru zasílat návrhy na dosud neuvedená rozšíření, která byste chtěli navíc implementovat. Zadavatel projektu rozhodne o přijetí/nepřijetí rozšíření a hodnocení rozšíření dle jeho ná-

ročnosti včetně přiřazení unikátního identifikátoru. Body za implementovaná rozšíření se počítají do bodů za projekt, takže stále platí získatelné maximum 25 bodů.

8.4.1 Některá rozšíření, která budou oceněna prémiovými body

Následující popisy rozšíření jazyka VYPe15 vždy začínají identifikátorem rozšíření a končí jejich bodovým ohodnocením.

- SHORTEVAL: Vyhodnocování podmínek v podmíněném výrazu a cyklu provádějte metodou zkráceného vyhodnocování (angl. *short evaluation*) (+1,5 bodu).
- GLOBAL: Podpora deklarací globálních proměnných (+1 bod)
- MINUS: Překladač bude pracovat i s prefixovým operátorem unární mínus a unární plus (priorita 8). Do dokumentace je potřeba uvést, jak je tento problém řešen (+1 bod).
- FOR: Překladač bude podporovat i cyklus **`for`** (+1 bod).
- HEAP: Implementace chybějící správy paměti typu halda s podporou nejen automatické alokace ale i dealokace. V dokumentaci popište vaše řešení (+3 body).
- IFONLY: Zjednodušený podmíněný příkaz **`if`** bez části **`else`**. Navíc místo bloku se může v podmíněném příkazu a příkazu cyklu vyskytovat pouze jeden příkaz. Do dokumentace je potřeba uvést, jak je tento problém řešen[7] (+1,5 bodu).
- INITVAR: Inicializace proměnné výrazem v rámci její definice (+0,5 bodu).
- OVERLOAD: Přetěžování uživatelských funkcí (+2 body).
- UNSIGNED: Zavedení bezznaménkových typů **`unsigned int`** (a v případě implementace rozšíření SHORT i **`unsigned short`**) včetně konverzí (bez kontroly přetečení) (+1 bod)
- BITOP: Podpora bitových operací **`~`**, **`&`** a **`|`** ve výrazech (+0,5 bodu).
- LOOP: Podpora příkazů **`break`** a **`continue`** se sémantikou analogickou jazyku C (+0,5 bodu).
- SHORT: Podporova menšího celočíselného typu **`short`** (16-bitové celé číslo). Explicitní přetypování **`short`** na **`int`** se provádí rozšířením znaménkového 16-bitového čísla na 32-bitové. V rámci implicitních konverzí je možné použít hodnotu typu **`short`** kdekoli je očekáván typ **`int`** (provede rozšíření se zachováním znaménka). Lze využít speciální instrukci **`READ_SHORT`** $*r*, která načte hodnotu čísla, ořízne na 16-bitů a uloží ji do registru *r*. Další instrukce **`PRINT_SHORT`** $*r* provádí výpis 16-bitovou hodnotu registru *r* jako číslo (+1 bod).
- ...

[7] Například, jak řešit u zanoření **`if`** a **`if`**-**`else`** párování **`if`** a **`else`**? U moderních programovacích jazyků platí konvence párování **`else`** s nejbližším **`if`**.