Zadání projektu z předmětu VYP

Luboš Lorenc
email: lorenc@fit.vutbr.cz

5. října 2005

1 Obecné informace

Název projektu: Implementace překladače imperativního jazyka *VYP05*.

Konzultace:

- Ing. Luboš Lorenc (vhodné si dopředu přes e-mail zarezervovat)
  - pondělí od 13:15 do 14:50
  - po dohodě ve výjimečných případech i jindy
- Ing. Rudolf Schönecker (vhodné si dopředu přes e-mail zarezervovat)
  - středa od 9:00 do 11:00
  - po dohodě ve výjimečných případech i jindy

Datum a způsob odevzdání: Do 22. prosince 2005 do 13:00, pouze prostřednictvím IS FIT.

Hodnocení:

- Maximálně 25 bodů (20 programová část, 5 dokumentace).
- Dokumentace je k programu, sama o sobě maximálně 1 bod.
- Za tvůrčí přístup (různá rozšíření a podobně) lze získat body navíc, maximálně však 25%.

2 Zadání

Definujte níže popsaný programovací jazyk *VYP05* vhodnou gramatikou. Na základě této gramatiky vytvořte syntaxí řízený překladač, který tento jazyk přeloží do strojového pseudokódu tak, aby byl spustitelný v dodaném interpretu.

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 regulárního výrazu. Klíčová slova jsou uzavřena v apostrofech, přičemž znak apostrof není součástí jazyka. Slovo `id` vždy zastupuje libovolný identifikátor. Bližší popis regulárních výrazů lze nalézt například v systému GNU/Linux po zadání příkazu `info grep`.

2.1 Popis programovacího jazyka

2.1.1 Obecné vlastnosti a datové typy

Jedná se o jednoduchý imperativní jazyk se dvěma základními datovými typy.

- Rozlišují se velká a malá písmena.
- Identifikátor **id** je definován regulárním výrazem `[a-zA-Z][a-zA-Z0-9_]*`
- Celočíselný literál **int** (konstanta) je definován výrazem `[0-9]+`
- Desetinný literál **float** (dvojnásobná přesnost) je definován regulárním výrazem `[0-9]+\.[0-9]*(e[+-]?[0-9]+)?`
- Datové typy pro jednotlivé uvedené literály jsou označeny klíčovými slovy
  - **int**
  - **float**
- Dále existuje datový typ **void**, který se používá pouze ve spojitosti s typem výsledku funkce a označuje typ prázdný, tedy, že se jedná o proceduru.

2.1.2 Definice funkcí

Jazyk umožňuje definici funkcí a procedur (pouze na globální úrovni) takto:

```
typ id ( \(  parametry \) )? [{] tělo [}]
```

Deklarace jednotlivých parametrů jsou odděleny středníkem, poslední deklarace není ukončena středníkem. Součástí každé deklarace může být klíčové slovo `'ref'`, udávající, že proměnná je předávána odkazem (implicitně jsou parametry předávány hodnotou). Nepovinná deklarace parametrů je tedy popsána následujícím regulárním výrazem:

```
typ ('ref')? id  (; typ ('ref')? id)*
```

Tělo funkce obsahuje vlastní algoritmus zapsaný takto:

```
(stmt)*
```

Jednotlivé příkazy jazyka `stmt` potom obsahují následující jazykové konstrukce:

- **Deklarace proměnných:** `'var' dcl (, dcl)* ;`
  - Jednotlivé `dcl` vypadají takto: `id (\[count\])? (, id (\[count\])?)* : typ` , kde `typ` zastupuje jeden ze dvou možných datových typů a `count` počet prvků v případě, že se jedná o deklaraci pole. Minimální velikost pole je jeden prvek. Pole jsou vždy indexována od nuly (první prvek má index 0). V případě, že se nejedná o deklaraci pole, nesmí být část `\[count\]` uvedena.
  - Proměnné je možné deklarovat v libovolném pořadí. Redeklarace nejsou povoleny — při opětovné deklaraci již deklarované proměnné dochází k sémantické chybě.
- **Podmíněný příkaz:**
  `'if' \( výraz \) [{] (stmt)* [}] ('else' [{] (stmt)* [}])? 'fi'`
  - Podmínka je splněna, pokud je výsledek výrazu nenulový.
  - Pro logické spojky ve výrazu generujte kód vyhodnocující metodou *short evaluation*.

- **Cyklus:** `'while' \( výraz \) 'do' [{] (stmt)* [}] 'done'`
  - Pro `výraz` platí stejné podmínky jako u podmíněného příkazu.
  - Sémantika této konstrukce je shodná se sémantikou konstrukce *while* v jazyce C/C++.
- **Volání funkce:** `id \( (výraz (, výraz)*)? \) ;`
  - Typy a počet skutečných parametrů musí odpovídat už definované funkci (v případě funkce bez parametrů se použijí prázdné závorky).
  - Pokud funkce vrací návratovou hodnotu, bude tato hodnota ignorována.
  - Pro logické spojky ve výrazech generujte kód vyhodnocující metodou *short evaluation*.
  - Funkce lze volat rekurzivně do libovolné úrovně zanoření.
  - U parametrů předávaných odkazem se jako prametr zadává absolutní adresa proměnné (kompilátor neprovádí žádný přepočet adres).
- **Přiřazení:** `id := výraz ;`
  - Identifikátor musí již být deklarován a výraz musí být odpovídajícího typu.
  - Pro logické spojky generujte kód vyhodnocující metodou *short evaluation*.
- **Návrat hodnoty z funkce:** `'ret' výraz ;`
  - Výraz musí být stejného typu jakého je funkce.
  - Pro logické spojky ve výrazu generujte kód vyhodnocující metodou *short evaluation*.

Každý program musí obsahovat funkci `main`, která představuje hlavní tělo programu. Tato funkce nemá parametry a nevrací výsledek. Funkce `main` může být definována kdekoliv ve zdrojovém souboru, ale pokud nezavedete do překladače rozšíření, všechny deklarace a definice za ní jsou prakticky nedostupné. Pokud však případ, kdy se vyskytuje nějaký text za funkcí `main`, nastane, je vhodné uživatele informovat případným chybovým hlášením.

Návratovou hodnotu funkce lze ignorovat. Každou funkci tudíž lze použít jako proceduru, aniž by se generovalo chybové hlášení (viz příkaz volání funkce).

2.1.3 Deklarace globálních proměnných

Jayzyk *VYP05* implicitně neobsahuje podporu pro práci s globálními proměnnými. Pokud však chcete, můžete jejich podporu provést jako rozšíření podle následujících pravidel.

Globální proměnné je možné deklarovat na libovolném místě programu (mimo těla fukcí), viditelnost je však až od místa deklarace. Deklarace globální proměnné vypadá následovně:

```
id (\[count\])? (, id (\[count\])?)* : typ ;
```

Význam jednotlivých částí deklarace je stejný jako v případě deklarace lokálních proměnných. Proměnné je opět možné deklarovat v libovolném pořadí. Redeklarace nejsou povoleny — při opětovné deklaraci již deklarované proměnné dochází k sémantické chybě.

2.1.4 Výrazy

Výrazy mohou obsahovat celá a desetinná čísla, odkazy na proměnné, volání funkcí (i rekurzivní) a níže uvedené operátory a funkce. Binární aritmetické operátory a binární logické spojky jsou zleva asociativní. Překladač musí podporovat implicitní přetypování výrazů typu **int** na typ **float**. Unární operátory $@, \#i$ a $\#f$ jsou zprava asociativní.

+, −, ∗, /	T x T → T, T je int nebo float. Operace / je pro celočíselné operandy definována jako celočíselné dělení.
%	int x int → int (modulo).
=, <>, >, <, >=, =<	T x T → int, T je int nebo float.
@	Unární prefixový operátor. T → int, T je libovolný typ. Zjištění absolutní adresy operandu v paměti (reference). V případě pole tento operátor vrací bázovou adresu pole. Jako rozšíření lze implementovat zjišťování adresy libovolného prvku pole.
$\#i$	Unární prefixový operátor. int → int. Obsah paměti na dané adrese interpretuje jako číslo typu **int**.
$\#f$	Unární prefixový operátor. int → float. Obsah paměti na dané adrese interpretuje jako číslo typu **float**.
and, or	T x T → int, T je int nebo float.
not	T → int, T je int nebo float.
readint()	Bez parametru → int. Čte číslo typu int ze standardního vstupu.
readfloat()	Bez parametru → float. Čte číslo typu float ze standardního vstupu.
printint()	int → bez návratové hodnoty. Zapíše parametr int na standardní výstup.
printfloat()	float → bez návratové hodnoty. Zapíše parametr float na standardní výstup.

Výraz může obsahovat i závorky a unární minus. Priorita operátorů je následující (nejvyšší na začátku):

()	závorky, volání funkce
$@, \#i, \#f$	
−	unární minus
∗, /, %	
+, −	
>, >=, <, =<	
=, <>	
not	
and	
or	

Poznámka k významu operátorů @, $\#i$ a $\#f$. Tyto operátory žádným způsobem nezkoumají význam odkazovaných dat. Bude-li například `x` proměnná typu **int**, potom příkaz `x:=#i123` způsobí uložení obsahu paměti od adresy 123 do adresy 126 včetně byte po bytu do proměnné `x` bez ohledu na to, zda tím vznikne nějaké smysluplné číslo typu int, či nikoliv. Budou-li `x` a `y` proměnné typu **int**, pak zápis `x:=#i@y` způsobí uložení obsahu proměnné `y` do proměnné `x`.

2.1.5 Poznámky k implementaci

Návrh a realizaci kompilátoru můžete provést libovolným způsobem. Vzhledem k rozsáhlosti kompilátoru a současným trendům v oblasti kompilátorů doporučuji maximální využití automatizovaných nástrojů pro podporu tvorby kompilátorů. Zejména se jedná o využití nástrojů **lex** nebo **flex** pro tvorbu scanneru a **bison** či **yacc** pro tvorbu parseru využívajícího LALR analýzu.

Podpora pro funkce readint, readfloat, printint a printfloat je již implementována v dodaném interpretu pomocí speciálních instrukcí (viz kapitola 2.2.2). Je pouze třeba zařídit správné provázání na úrovni generovaného kódu.

Veškerá chybová a varovná hlášení vzniklá v průběhu překladu vypisujte na **standardní chybový výstup**. Samozřejmostí je, že na tento výstup nebude vypisováno nic jiného než varování a chybová hlášení. *Nedodržení této podmínky se hodnotí jako neodevzdaný projekt!*

2.2 Cílový strojový pseudokód a formát cílového souboru

Cílový soubor musí odpovídat následujícímu pevnému formátu.

Soubor je načítán po řádcích. Jednotlivá slova mohou být oddělena libovolným oddělovačem, doporučuji však používat mezeru. Za slovo je považován blok textu oddělený z obou stran jedním nebo více oddělovači. Za oddělovače jsou považovány standardní oddělovací znaky, jako je mezera, tabelátor a podobně (přesný seznam lze najít v dokumentaci ke standardní funkci jazyka C *isspace*()). Před prvním a za posledním slovem na řádku nemusí být žádný oddělovač.

2.2.1 Popis cílového souboru

- Cílový soubor je tvořen posloupností instrukcí. Na každém řádku je právě jedna instrukce.
- Každý řádek, který začíná znakem % je považován za komentář a při interpretaci bude přeskočen. Stejně tak za každou instrukcí může být uveden komentář bez jakéhokoliv uvozování (interpret z každého řádku načte právě tolik slov, kolik potřebuje k vykonání dané instrukce a vše ostatní ignoruje).
- Interpret umožňuje specifikovat přechod do interaktivního ladicího režimu. Ten se zapíná v okamžiku načtení řádku tvořeného jediným znakem @.

2.2.2 Popis cílového procesoru a jeho instrukcí

Cílový procesor obsahuje:

- 16 univerzálních registrů číslovaných 0–15,
- čítač programu (registr PC),
- bázový registr pro adresaci (registr BP),
- paměť organizovaná po bytech, velikost definovaná při startu interpretu,
- zásobník organizovaný po bytech, velikost definovaná při startu interpretu,
- podporu pro trasování programu.

Dále 'X' označuje jedno z písmen I nebo F (pro operandy typu **int** či **float**).

- **Aritmetické operace**
  - $x + y \rightarrow z$; x, y, z jsou čísla registrů
    `ADDX x y z`
  - $x - y \rightarrow z$; x, y, z jsou čísla registrů
    `SUBX x y z`
  - $x * y \rightarrow z$; x, y, z jsou čísla registrů
    `MULX x y z`

- $x/y \rightarrow z$; x, y, z jsou čísla registrů (pro I je to celočíselné dělení)
  `DIVX x y z`
- $x\%y \rightarrow z$; x, y, z jsou čísla registrů (celočíselné modulo)
  `MOD x y z`

- **Operace porovnání** — u těchto operací platí, že první dva operandy mohou být libovolného (ale stejného) typu a výsledný operand (registr) je typu **int**.
  - $x > y \rightarrow z$; x, y, z jsou čísla registrů
    `GTX x y z`
  - $x >= y \rightarrow z$; x, y, z jsou čísla registrů
    `GEX x y z`
  - $x < y \rightarrow z$; x, y, z jsou čísla registrů
    `LTX x y z`
  - $x <= y \rightarrow z$; x, y, z jsou čísla registrů
    `LEX x y z`
  - $x == y \rightarrow z$; x, y, z jsou čísla registrů
    `EQX x y z`
  - $x <> y \rightarrow z$; x, y, z jsou čísla registrů
    `NEX x y z`

- **Test nulové hodnoty** — tato operace může mít první operand libovolného typu, výsledek je vždy typu int.
  - $x == 0 \rightarrow y$; x, y jsou čísla registrů
    `ISZX x y`

- **Práce s pamětí**
  - $mem(x) \rightarrow y$ — obsah paměti z adresy dané obsahem registru x uloží do registru y. Pokud je místo čísla registru x uvedeno libovolné číslo předcházené znakem #, jedná se o přímé adresování (uvedené číslo přímo udává adresu paměťové buňky).
    `LDDX x y`
  - $mem(x + BP) \rightarrow y$ — obsah paměti z adresy dané obsahem registru x inkrementovaným o obsah registru BP uloží do registru y. Pokud je místo čísla registru x uvedeno libovolné číslo předcházené znakem #, jedná se o přímé adresování (uvedené číslo po inkrementaci o obsah registru BP přímo udává adresu paměťové buňky).
    `LDBX x y`
  - $x \rightarrow mem(y)$ — obsah registru x uloží do paměti. Výpočet adresy probíhá stejně jako v případě instrukce LDDX.
    `STDX x y`
  - $x \rightarrow mem(y + BP)$ — obsah registru x uloží do paměti. Výpočet adresy probíhá stejně jako v případě instrukce LDBX.
    `STBX x y`

  Tyto instrukce vždy zapisují či čtou správný počet bytů paměti podle typu instrukce. Například pro celočíselné operace to jsou 4 byty. Ale pozor — zadávané adresy nejsou násobeny velikostí operandů!

- **Práce s čítačem programu**

  - $PC \rightarrow x$ — obsah čítače programu (právě prováděnou instrukci) uloží do registru x. Teprve poté se provede inkrementace PC.
    `LDPC x`

- **Práce s konstantami**
  - $inum \rightarrow reg$; reg je číslo registru, $inum$ je dekadická konstanta.
    `LDCI inum reg`
  - $fnum \rightarrow reg$; reg je číslo registru, $fnum$ je desetinná konstanta ve formátu `-?[0-9]+(\.[0-9]*)?(e-?[0-9]+)?`
    `LDCF fnum reg`

- **Operace přesunu**
  - $reg1 \rightarrow reg2$; $regx$ je číslo registru
    `MOV reg1 reg2`

- **Typové konverze**
  - $x \rightarrow y$; x je číslo registru obsahujícího hodnotu typu **int**, y je číslo registru, do něhož bude uložena tatáž hodnota zkonvertovaná na typ **float**.
    `I2F x y`

- **Skokové instrukce** — pro interpret představuje celý program nedělitelný blok instrukcí číslovaný od 0. Skoky jsou vždy absolutní (JMP 10 skočí na 11. instrukci od začátku programu, JMP 0 na první, ...).
  - skok na instrukci danou indexem při nenulovém registru
    `JTX reg instr`
  - skok na instrukci danou indexem při nulovém registru
    `JFX reg instr`
  - skok na instrukci danou indexem
    `JMP instr`

- **Instrukce pro práci se zásobníkem** — tyto instrukce zapisují (či vybírají) na zásobník právě tolik bytů, jaká je velikost operandu (například pro **int** to jsou 4 byty).
  - $reg \rightarrow stack$; reg je číslo registru
    `PUSHX reg`
  - $stack \rightarrow reg$; reg je číslo registru
    `POPX reg`

- **Volání podprogramů**
  - volání podprogramu (funkce)
    `CALL addr`
    Na zásobník je uložena hodnota PC+1 (aby běh programu po návratu mohl pokračovat příští instrukcí) a poté je PC nastaven na hodnotu `addr` — konstanta `addr` musí být adresou první instrukce volaného podprogramu. Při volání podprogramů nezapoměňte, že interpret za Vás neudělá stack frame. Zároveň je vhodné či spíše nutné postarat se o zápis správné nové bázové adresy pro addresování do registru BP.
  - návrat z procedury nebo funkce.
    `RET`
    Vyjme ze zásobníku nejvrchnější hodnotu a zapíše ji do PC. Pokud je z podprogramu

potřeba vrátit návratovou hodnotu na zásobníku, musí si tento podprogram zajistit její korektní uložení (vyjmout uložený obsah PC, uložit návratovou hodnotu, opět na vrchol vložit obsah PC). Opět je nutné, postarat se o stack frame.

- **Pseudoinstrukce pro vstupně-výstupní operace**
  - *reg* → *stdout*; *reg* je číslo registru. Vypíše obsah registru na standardní výstup jako **int**.
    `PRINTINT reg`
  - *reg* → *stdout*; *reg* je číslo registru. Vypíše obsah registru na standardní výstup jako **float**.
    `PRINTFLOAT reg`
  - *stdin* → *reg*; *reg* je číslo registru. Načte ze standardního vstupu číslo typu **int** do registru.
    `READINT reg`
  - *stdin* → *reg*; *reg* je číslo registru. Načte ze standardního vstupu číslo typu **float** do registru.
    `READFLOAT reg`

3 Instrukce ke způsobu vypracování a odevzdání

3.1 Obecné informace

Všechny odevzdávané soubory budou programem TAR+GZIP či ZIP zkomprimovány do jediného archivu, který se bude jmenovat přihlašovací_jméno.tgz, či přihlašovací_jméno.zip. Archiv nebude obsahovat adresářovou strukturu a speciální soubory. Všechny názvy souborů uvnitř archivu budou náležet do jazyka definovaného tímto regulárním výrazem `[a-z_.0-9]+`. Řešení budou zpracována automatizovanými prostředky a při nedodržení těchto pokynů nebude řešení zpracováno a bude považováno za neodevzdané. Archivní soubor bude odevzdán přes IS FIT.

Dále je třeba odevzdat celý projekt v daném termínu (viz výše). Pokud tomu tak nebude, je projekt opět považován za neodevzdaný, nebo minimálně se dá očekávat ztráta bodů dle toho, o jaké zpoždění půjde. Stejně tak pokud se bude jednat o plagiátorství jakéhokoliv druhu, je projekt považován za neodevzdaný.

3.2 Požadavky na řešení

3.2.1 Textová část řešení

Součástí řešení bude dokumentace, vypracovaná ve formátu PDF či PS — jiný formát je nepřípustný. Dokumentace bude vypracována v **anglickém jazyce**. Tato dokumentace bude obsahovat:

- Jméno, příjmení a přihlašovací jméno řešitele,
- popis vašeho způsobu řešení — cca 2–4 strany (obrázky se do délky nezapočítávají),
- upozornění na případná rozšíření proti zadání — dle implementace.

Dokumentace nesmí obsahovat:

- Kopii zadání,
- text, který není Váš původní (kopie z přednášek, sítě, WWW, ...)

Dokumentace bude uložena v jednom souboru se jménem přihlašovací_jméno.XXX, kde XXX je buďto **pdf** nebo **ps**. Jakékoliv jiné formáty dokumentace, než předepsané, jsou ignorovány, což vede ke ztrátě bodů za dokumentaci.

3.2.2 Programová část řešení

Programová část řešení bude vypracována v jazyce (ANSI) C/C++. Programy musí být přeložitelné překladačem gcc/g++. Při hodnocení budou projekty překládány pod OS GNU/Linux. 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, ztrácíte právo na reklamaci výsledků. Ve sporných případech bude vždy za platný považován výsledek překladu na serveru merlin.

Součástí řešení bude soubor přihlašovací_jméno**.mak** nahrazující soubor Makefile (projekty budou překládány pomocí `gmake -f přihlašovací_jméno.mak`[1]).

Binární soubor (přeložený kompilátor) v žádném případě neodevzdávejte v archivu (jednak bude archiv zbytečně velký a binární soubor bude stejně okamžitě smazán, navíc budete postihnuti ztrátou bodů!). Pokud soubor pro sestavení cílového programu nebude obsažen, nebo se na jeho základě nepodaří sestavit cílový program, hodnotí se projekt jako neodevzdaný. Připomínám, že vše musí být přeložitelné z prostředí sítě FIT **bez** dodatečných nastavení a pouze v uvedených typech překladačů.

První tři řádky zdrojových textů a dokumentace musí obsahovat název projektu, přihlašovací jméno studenta a jméno studenta. Pokud tomu tak nebude, je projekt hodnocen automaticky 0 body.

Program převezme z příkazové řádky jméno vstupního souboru. Výstup programu bude uložen do souboru stejného jména bez poslední přípony, ale s novou příponou **.out** a do stejného adresáře, ze kterého pochází vstupní soubor. V případě, že jméno vstupního souboru nebude obsahovat příponu, bude pouze přidána přípona **.out**. Například tedy bude volán překlad `kompiluj Testy/test01.vyp` a ten vytvoří v adresáři *Testy* soubor *test01.out*. Nedodržení se hodnotí jako neodevzdaný projekt.

Veškerá chybová hlášení generovaná kompilátorem budou vždy vypisována na **standardní chybový výstup**. Kromě chybových hlášení nebude kompilátor v průběhu překladu na žádný výstup vypisovat žádné texty. Stejně tak přeložený program nebude vypisovat nic jiného než to, co bylo naprogramováno ve zdrojovém programu! Projekty budou opravovány pomocí automatu využívajícího sadu testovacích příkladů a mimo jiné i program diff. Vypsání i jediného neočekávaného znaku pravděpodobně způsobí nesprávné vyhodnocení Vašeho výstupu a minimálně ztrátu bodů. *Obecně je ale možné nedodržení těchto podmínek hodnotit jako neodevzdaný projekt!!!*

3.3 Konzultace

Konzultace budou probíhat ve výše uvedených termínech a ve výjimečných případech i jindy. Doporučuji se na ně předem domluvit. Domluva předem pomůže řešit případný zájem více jedinců o konzultaci ve stejný čas a taktéž se takto ujistíte o připravenosti a přítomnosti cvičícího. Pokud vyžadujete konzultaci, tak na ni, prosím, choďte připraveni. Projekt není triviální. Budete-li proto mít kdykoliv pocit, že Vám není něco jasné nebo že je někde chyba, neváhejte a ihned mě kontaktujte e-mailem. Na pozdní „nářky“ nelze brát zřetel.

3.4 Některá doporučená rozšíření

- přidělování registrů
- další optimalizace
  - constant folding

[1] Makefile je soubor, který zpracovává program make. V žádném případě to není shellový skript! Více informací lze najít v `info make` nebo na http://www.gnu.org. (Soubor Makefile tedy pouze přejmenujte na přihlašovací_jméno**.mak**)

  - řetězení v SHORT-EVALUATION
  - optimalizace cyklů
  - eliminace mrtvého kódu
  - ...
- další aritmetické či logické operátory s pravou asociativitou (např. mocnina)
- globální proměnné
- další programové konstrukce
  - cyklus typu *do — while*
  - cyklus typu *for*
  - příkaz násobného větvení typu *case – of*
- komentáře (triviální rozšíření)

3.5 Zásady bodového hodnocení

Hlavní zásady pro vyhodnocení odevzdaného projektu:

- Na to, aby projekt mohl být vůbec hodnocen, musí generovat výstup, který odpovídá alespoň tomu nejjednoduššímu zadání a interpret je schopen ho správně interpretovat.
- Pokud je dodána pouze dokumentace, tak ta je hodnocena maximálně jedním (1) bodem.
- Pokud program negeneruje výstup nebo výstup není korektní, či neodpovídá vstupnímu programu, je programová část projektu hodnocena maximálně dvěma (2) body.
- Opravování bude provedeno testováním kompilátoru na sadě testovacích příkladů (zdrojových testů). Každý přeložený příklad bude spouštěn v interpretu a jeho výstup porovnán se správným výstupem. Za každý správně přeložený testovací příklad bude přidělen odpovídající počet bodů.

3.5.1 Projekt je hodnocen jako neodevzdaný, t.j. 0 body, pokud

- Nebude odevzdán včas (za každý započatý den zpoždění minimálně 20% bodů dolů),
- nebude odevzdán v předepsaném formátu,
- nebude možné sestavit program tak, jak je uvedeno v zadání (např. je nutný další zásah do předpisů pro sestavení, ruční nastavení přístupových cest, ...),
- půjde o plagiát v jakékoliv podobě,
- výstup programu nebude odpovídat zadání,
- ...

3.6 Jak postupovat při řešení projektu

Při řešení je pochopitelně možné využít vlastní výpočetní techniku. Instalace překladače gcc/g++ není nezbytně nutná, pokud máte jiný překladač C/C++ již instalován a nehodláte využívat vlastnosti, které právě tento má a žádný jiný překladač nemá. Při použití pouze jazyka C by problém neměl nastat vůbec, u C++ je dobré si před použitím nějaké *vyspělé* konstrukce ověřit, že jí disponuje i překladač, který nakonec bude použit při opravování. Po vypracování je ale vhodné vše ověřit na cílovém překladači, aby při kontrole vše proběhlo bez problémů.

Pro vlastní řešení je dobré postupovat tak, aby Váš produkt co nejdříve generoval funkční a správný kód pro nějaké jednoduché programové konstrukce. Tj. něco, co například dokáže přeložit součet dvou čísel a tisknout výsledek. To totiž umožní hodnotit projekt jako celek a přiznat více, než jen symbolické body.

Takový přístup je možné například realizovat tak, že nejdříve uděláte lexikální analyzátor, poté navrhnete gramatiku a na jejím základě uděláte syntaktický analyzátor, který nejdříve umí jen definice funkcí a tisk výrazů s konstantami. Nad tímto ale již implementujete generátor. A tak máte alespoň mikropřekladač hotový, což už na něco stačí.

Poté přidáváte další syntaktické konstrukce a s tím přidáváte i další prvky do generátoru a postupně budujete i sémantickou analýzu.

Doporučuji ponechat si dostatek času pro implementaci alespoň některých rozšíření. Často stačí drobná chyba z nepozornosti na to, aby způsobila havárii kompilátoru na velkém množství testovacích příkladů. V tom případě je bodové hodnocení často velmi neadekvátní vynaložené námaze. Přitom právě implementace doporučených rozšíření je faktem, který svědčí o Vašem zájmu o danou problematiku. Pokud přijdete reklamovat hodnocení rozsáhlého a propracovaného kompilátoru s několika rozšířeními, budete mít určitě větší šanci na úspěch, než když bude konečná diskuze probíhat nad jednoduchým kódem plným chyb, zcela evidentně psaným ve spěchu bez dostatečných znalostí.

3.7 Doplňující podmínky

Opravování projektů bude prováděno na nejnovějším překladači gcc, který bude v době hodnocení k dispozici. V současné době se jedná o verzi gcc 3.4.4. V době opravování projektu to ale pravděpodobně bude některá z verzí 4.0.x.

3.8 Stanovisko k akademické nečestnosti

Pokud se při vytváření projektu dopustíte jakéhokoliv přestupku proti akademické morálce (např. plagiátorství), Váš přestupek bude ohlášen garantovi předmětu, který je současně předseda Disciplinární komise FIT VUT. Takovýto podvod typicky vyústí v neudělení zápočtu a tím k neúspěšnému absolvování předmětu VYP; výslovně však upozorňujeme, že může vést k vyloučení z akademické obce a tím pádem z celého VUT.

Obsah