VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ROZŠÍŘENÍ PROGRAMOVACÍHO JAZYKA C A JEHO PŘEKLADAČ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR OPATŘIL
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ROZŠÍŘENÍ PROGRAMOVACÍHO JAZYKA C A JEHO PŘEKLADAČ

EXTRENSION OF THE C PROGRAMMING LANGUAGE AND ITS COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR OPATŘIL
AUTHOR

VEDOUCÍ PRÁCE Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Tématem bakalářské práce je návrh a realizace nového programovacího jazyka C+, implementovaného jako rozšíření programovacího jazyka C. Mimo to je provedeno srovnání vybraných populárních jazyků a vysvětleny přednosti a smysl rozšíření, dále je teoreticky i konkrétně vysvětlen princip překladače.

Abstract

The topic of this thesis is designing and realization of new programming language C+, implemented as extension of programming language C. Aside from that, comparison of several popular languages was conducted in order to discuss advantages and concept of the extension. Workings of compiler is explained, both theoretically and practically.

Klíčová slova

rozšíření C, jazyk C, překladače, C+

Keywords

C extension, C language, compilers, C+

Citace

Petr Opatřil: Rozšíření programovacího jazyka C a jeho překladač, bakalářská práce, Brno, FIT VUT v Brně, 2015

Rozšíření programovacího jazyka C a jeho překladač

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. RNDr. Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Petr Opatřil
18. května 2015

Poděkování

Rád bych poděkoval svému vedoucímu, panu prof. Medunovi, za velmi příjemnou spolupráci.

© Petr Opatřil, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Analýza jazyků **4**
2.1 Jazyk symbolických instrukcí 4
2.2 Jazyk C 5
2.3 Jazyk C++ 7
2.4 Python 7
2.5 Ostatní 7

3 Koncepce jazyka C+ **8**

4 Teorie jazyků a překladu **11**
4.1 Formální jazyky a gramatiky 11
4.2 Bezkontextové gramatiky 12
4.3 Lexikální analýza 13
4.4 Schéma překladače 13
4.5 Tabulka symbolů 14
4.6 LL a LR analýza, rekurzivní sestup 14
4.7 Konverze LR do LL 15
4.8 Gramatika C+ 16

5 Rozšíření v jazyku C+ **18**
5.1 Odsazování bílými znaky 18
5.2 Rozšíření řídících konstrukcí 20
5.3 Implicitní argumenty 22
5.4 Vlastní datové typy 22
5.5 Přetěžování funkcí 24
5.6 Volnější syntaxe 27

6 Implementace **30**

7 Závěr **32**

A Obsah CD **36**

Seznam obrázků

2.1 C selekce přepsaná do pseudoassembleru 5
2.2 C cyklus přepsaný do pseudoassembleru 6
2.3 Příklad vyznamu ukazatelů: seznam čísel 6

3.1 Ukázka snadné kanonické konverze některých rozšíření C+ 10

4.1 Příklad bezkontextové (LR) gramatiky, popisující výrazy 12
4.2 Příklad nejednoznačné gramatiky 12
4.3 Ukázka lexikální analýzy 13
4.4 Příklad LL gramatiky 14
4.5 Dva typy konverzí při překladu z LR do LL 15
4.6 Ilustrace problémů při překladu z LR do LL a jejich řešení 16

5.1 Ukázka odsazování bílými znaky 19
5.2 Využití strukturované konstrukce `switch` 21
5.3 Vhodné případy použití implicitních argumentů 22
5.4 Vhodné typy pro přetížení operátorů 23
5.5 Příklady přetěžování operátorů 24
5.6 Porovnání způsobů zakódování jmen mezi překladači[8] 25
5.7 Příklady přetížených funkcí s aplikovanými zakódovanými jmény 26
5.8 Hodnotící funkce srovnávající konverzi z typu ve volání funkce na typ v daném přetížení 26
5.9 Vzorek kódu s absencí středníků 27
5.10 Příklady nejasného kódu při uvolnění syntaxe 28

Kapitola 1

Úvod

Základní myšlenkou, stojící za návrhem na tento projekt, bylo zamyšlení se nad popularitou jazyka Python. Ten je často ceněn pro velkou čítelnost a kompaktnost zápisu, vlastnosti typické pro vyšší programovací jazyky. Cenou za abstrakci a komfort zápisu je však slabší výkon a vyšší nároky na paměť oproti takovému C, inherentní pro interpretovaný jazyk. Proto je Python a podobní často využíván jednak pro krátké, jednoduché skripty (kde není výkon podstatný či jsou na jedno použítí), nebo jako tenká, řídící vrstva nad knihovnami v rychlejším, kompilovaném jazyce. Toto poznání budí otázku, zda by nebylo možné zmíněné výhody vyšších jazyků reprodukovat v nižším jazyce bez ztráty efektivity.

Tématem práce je návrh právě takového jazyka, kombinující optimalizovanost nízkoúrovňového jazyka s čitelností a elegancí jazyka vyššího. Z praktických důvodů, rozvedených v kapitole 3, je výhodné, aby tento jazyk byl zpětně kompatibilním rozšířením jazyka C. V rámci návrhu byla provedena kritická analýza některých populárních programovacích jazyků s jejich klady a zápory jako zdroj inspirace. Dále byla navržena formální gramatika jazyka, v jejímž rámci byla LR gramatika mateřského C převedena do LL. Na závěr byl vytvořen překladač do čistého C.

Tato technická zpráva se obsahově drží zmíněných etap práce. V kapitole 2 je proveden rozbor jazyků a vysvětleny důležité pojmy, od čtenáře se však jistá znalost oboru očekává. V kapitole 3 je představen vytvořený jazyk a jeho filosofie, v kapitole 4 pak jeho gramatika spolu s vysvětlením spjaté teorie formálních jazyků a překladačů. Následující kapitoly se věnují konkrétním rozšířením jazyka, spolu s jejich významem, srovnáním s konkurencí a možností realizace. Kapitola 6 poskytuje nástin implementace překladače a konečně závěrečná kapitola 7 shrnuje výsledky a představuje možná budoucí témata práce.

Kapitola 2

Analýza jazyků

Při návrhu nového jazyka byla podrobně prozkoumána řada imperativních programovacích jazyků, zváženy jejich klíčové klady a zápory a zdůrazněna případná unikátní funkcionalita. Primárními zdroji byly standardy/reference jednotlivých jazyků a kniha [13].

2.1 Jazyk symbolických instrukcí

Prvním řešeným je jazyk symbolických instrukcí. Jedná se o nejnižší možnou úroveň abstrakce nad strojovým kódem – jednotlivé symbolické instrukce se přesně mapují na konkrétní instrukce procesoru. Z toho plyne nekolik faktů – programováním v tomto jazyku lze dosáhnout nejrychlejší a nejoptimalizovanější implementace daného kódu, zatímco efektivita ostatních kompilovaných jazyků závisí na kvalitě a optimalizačních schopnostech překladačů. Běžným trendem je, že čím je programovací jazyk abstaktnější a model výpočtu vzdálenější od (typicky Von Neumannovské) architektury počítače, tím komplikovanější je nutný překladač, produkující méně optimalizovaný výsledný kód[1]. Z principu jazyka je zřejmé, že v jazyce symbolických instrukcí lze naprogramovat libovolný program, jaký počítač dokáže spustit, což se nedá u žádného vyššího jazyka zaručit.

Poznámka. Výše zmíněné tvrzení chápe program jako posloupnost instrukcí, kdy překladač z vyššího jazyka nemusí být schopen využít specializované instrukce (kupř. VLIW,MMX) stejně efektivně jako programátor. Pokud je ale program chápán jako obecná výpočetní úloha, pak je teoretická výpočetní síla většiny programovacích jazyků srovnatelná, jsou všechny tzv. Turingovsky úplné, tedy odpovídají Turingovu stroji[10].

Přestože však obecně automatizované optimalizační techniky nemohou předčit ruční optimalizaci, tento jazyk má řadu nevýhod. Především je zde problém nepřenositelnosti, vycházející z toho, že různé procesory mají různé instrukční sady. Mimo to, ačkoli teoreticky lze programovat bez omezení, absence vyšších datových struktur a nestrukturovanost jazyka činí psaní rozsáhlejšího kódu velmi nepraktické. V praxi je tedy nejčastěji využíván pro krátké, vysoce zoptimalizované funkce – jazyky C a C++ za tímto účelem přímo dovolují jazyk symbolických instrukcí vnořovat do kódu[2].

Ve výsledku je tedy jazyk symbolických instrukcí velmi zajímavý svojí optimálností, které se navrhovaný jazyk bude snažit co nejvíce přiblížit, pro přímé využití však vhodný není.

[1] U jazyků interpretovaných je režie jestě vyšší, neboť kromě hardwarové interpretace instrukcí interpretu přibude softwarová mezivrstva interpretující program samotný.

[2] Ve skutečnosti jde pro C pouze o široce podporované rozšíření, byť popsané v nenormativní části standardu. V C++ je součástí jazyka, sémantika je ale implementačně definovaná.

2.2 Jazyk C

Pro práci na větším projektu je nutné kód vhodně strukturovat, jedním z nejpopulárnějších strukturovaných, modulárních jazyků je jazyk C. O jeho oblíbenosti svědčí i fakt, že z jeho gramatiky vychází celá řada dalších jazyků (mimo jiné všechny dále jmenované). Z hlediska výkonnostního je C velmi dobře optimalizovatelné, což plyne z faktu, že konstrukce a datové typy z C jsou v jazyce symbolických instrukcí triviálně realizovatelné.

```
if (5+3)
  {*b*;*c*;*d*;}
else
  {*e*;*f*;}
*g*;
```

$$\Rightarrow$$

```
  MOV registr, 5 //presun do registru cislo 5
  ADD registr, 3 //soucet registru a 3
  JNE navesti_else //skoc na navesti, pokud se posledni hodnota v registru
  nerovna 0
  *b* //instrukce odpovidajici příkazu *b*
  *c*
  *d*
  JMP navesti_konec //vzdy skoc na navesti
navesti_else:
  *e*
  *f*
navesti_konec:
  *g*
```

Obrázek 2.1: C selekce přepsaná do pseudoassembleru

Jazyk C je procedurální, kód programu je tedy prováděn voláním uzavřených podprogramů – funkcí. Modularita je docílena tím, že funkce a proměnné nemusí být definovány, ale mohou být pouze deklarovány, tedy pouze oznámena jejich existence, zatímco definice může být umístěna v jiném modulu.

Datové typy jsou statické, pro danou proměnnou určené při její deklaraci. Kromě celočíselných a desetinných typů je podporován datový typ struktura (složenina z více datových typů), typ unie (jedno místo v paměti reinpretovatelné jako jeden z více datových typů) a výčtový typ (celočíselný typ, kde jednotlivé hodnoty odpovídají speciálním konstantám). Toto tvoří základ typu, ten může být (vícenásobně) nepřímo odkazován ukazatelem[3], polem[4] nebo funkcí (funkčním ukazatelem) základ vracející. Funkce nemusí vracet žádný typ (pseudotyp `void`). Konstrukce `typedef` umožnuje pro konkrétní datový typ vytvořit synonymum, chovající se syntakticky jako nový datový typ.

Ačkoli C obsahuje příkaz nepodmíněného skoku `goto`, jeho strukturované řídící konstrukce – `if` a `switch` pro selekci a vícenásobnou selekci; `for`, `while` a `do-while` pro iteraci; rekurzivní volání funkcí – jsou dostatečné pro libovolnou výpočetní úlohu.

Součástí jazyka je i tzv. *preprocesor*, který lze chápat jako unikátní jazyk, jehož vyhodnocení probíha před vlastním překladem. Běžný C kód není nijak vyhodnocován, preprocesor

[3] skalární typ s adresou v paměti

[4] efektivně odpovídajícímu ukazateli na první prvek

```
for (i=0;i<5;i++)
{
  *a*;
}
*b*;
```

$\Rightarrow$

```
  MOV i, 0
navesti_cyklus:
  MOV registr, i
  SUB registr, 5 //odecti od registru 5
  JE navesti_konec_cyklu //skoc na navesti, pokud se posledni hodnota v
  registru rovna 0
  *a*
  INC i //inkrementuj i
  JMP navesti_cyklus
navesti_konec_cyklu:
  *b*
```

Obrázek 2.2: C cyklus přepsaný do pseudoassembleru

```
struct list
{
  struct list *next; // Ukazatel na nasledujici polozku seznamu
  int value;
};
```

Obrázek 2.3: Příklad vyznamu ukazatelů: seznam čísel

reaguje jen na vlastní direktivy, některé popsané dále: Directiva `include` umožnuje vložit na dané místo jiný soubor – typicky tzv. hlavičkový soubor informující o funkcích z jiných modulů. Direktiva `define` definuje makro, které je při použití substituované za jiný kód. Konečně direktiva `if` umožnuje podmíněný překlad části kódu.

Informace o jazyku C byly primárně čerpány ze standardu C99[4]. Pro shrnutí, C je kompaktní a velmi efektivní jazyk, který v rychlostních srovnáních často zabírá přední příčky[5] a ve kterém jsou často psány knihovny (významným příkladem je jádro Unixových operačních systémů[1]) nebo kód pro vestavěné systémy.

Nevýhody jazyka C plynou z jeho nízkoúrovňovosti. Práce s ukazateli vede snadno k chybám s nepovoleným přístupem do paměti, chybí strukturovaná práce s pamětí a ošetřování chyb. Obecně vzato, modelování abstraktnějších konceptů vede k dlouhému a málo čitelnému kódu, proto C není vhodné na velmi rozsáhlé, skupinové projekty.

[5] V některých matematických aplikacích býval tradičně porážen Fortranem, standard C99 přidal typový kvalifikátor `restrict` pro srovnatelnou optimalizaci.

2.3 Jazyk C++

Jazyk C++ lze chápat jako velké rozšíření jazyka C, nejde však o striktní nadmnožinu[6]. Je to jazyk multiparadigmový, zachovávající možnosti C a přidávající techniky objektově orientovaného programování. Mimo to obsahuje pokročilé možnosti šablonového metaprogramování a lambda funkce z programování funkcionálního[14].

Po stránce výkonnostní se C++ snaží blížit C a skutečně má vůči jazykům srovnatelné třídy (jako Java) vyšší výkon. Při ryze procedurálním kódu se výsledek od C prakticky neliší. Silné využívání objektových vlastností jazyka (jako dynamický polymorfismus) jistou režii má, přesto např. využití nově přidaných R-hodnotových referencí minimalizuje zbytečné kopírování, jinak častý problém neoptimálně napsaného OO kódu.

Základním principem objektového programování je svázání dat s manipulujícím kódem, to je realizováno přidáním funkcí do definic tříd (složený typ identický s C strukturou). Pomocí zapouzdření lze některé členy třídy znepřístupnit pro externí funkce. Společně s přetěžováním funkcí a operátorů (viz 5.5, 5.4), dědičností, implicitními argumenty (5.3) a obsluhou vyjímek lze psát čitelnější kód, který navíc jasně odděluje vnější rozhraní od vnitřní implementace. Pro zvýšení kompatibility nesouvisejících modulů zavádí také prostory jmen.

Stinné stránky vychází z rozsáhlosti jazyka – stát se expertem je velmi dlouhodobý úkol a psát neefektivní kód je snadné, mnoho funkcionality je duplikované (pole vs. knihovní typ vector, ukazatele a reference, makra a šablony), překlad je vůči C výrazně složitejší a tedy i pomalejší atd.

2.4 Python

Python je multiparadigmový interpretovaný jazyk, jehož hlavní předností je vysoká čitelnost. Přestože také vychází z C syntaxe, nemá středníky a používá unikátní odsazení bílými znaky (rozvedeno v 5.1, 5.6). Možnostmi se podobá C++ s několika rozdíly – členové tříd nemají jazykem omezenou viditelnost, přetížení operátorů je realizováno speciálně pojmenovanou metodou apod. – více viz [2].

Přetěžování funkcí není podporováno, to ale souvisí s faktem, že Python je dynamicky typovaný, datové typy se vážou na jednotlivé objekty a v proměnných jsou na ně uloženy reference. Tento systém je flexibilní, ale nepříliš výkonostně efektivní, což se dá jako popis vztáhnout na celý jazyk.

2.5 Ostatní

Kromě jmenovaných byly prozkoumány i další jazyky – GNU C, Objective-C, Php, C#, Java. Jejich vlastnosti a gramatiky byly porovnány s ostatními. Vlastnosti všude řešené stejně byly chápany jako potvzení dobrého návrhu, unikátnosti byly až na vyjímky jako řízení cyklů v Javě (viz víceúrovňový `break` v 5.2) buď horší než stávající řešení (např. $ u proměnných v Php) nebo nešly vhodně realizovat jako rozšíření C. Některé zajímavé pozorované vlastnosti (jako různé varianty cyklu `foreach` v jazycích C++, Python, Php, C#) nebyly řešeny z časových důvodů.

[6] C++ je např. mnohem striktnější v implicitních typových konverzích

Kapitola 3

Koncepce jazyka C+

Po analýze jazyků v předchozí kapitole lze formulovat na navrhovaný jazyk následující požadavky. Musí mít výkon nízkoúrovňového jazyka, přesto musí být portabilní. Jazyk C je na to vhodný kandidát, jeho problémem je ale rozsáhlost kódu při řešení větších projektů. Navrhovaný jazyk by se měl být schopen čitelností zápisu srovnávat s Pythonem či C++, aby měl však praktický význam, nesmí těmito rozšířeními utrpět výkonnost, resp. realizace rozšíření musí být nejvýše stejně komplikovaná, jako by bylo stejné řešení v C. Ukazuje se, že právě takových rozšíření, která zvýší čitelnost a jsou triviálně implementovatelná, je celá řada.

Vytvořit kompletně nový kompilovaný jazyk by byl zbytečně složitý úkol, výhodnější bylo vzít standardní C (ISO C99) a nové vlastnosti do něj inkrementálně přidávat. Zároveň to silně zjednodušuje problematiku překladače, neboť stačí přeložit rozšíření do čistého C a samotnou tvorbu překladu přenechat běžnému C kompilátoru, který se už postará o optimalizace a širokou multiplatformní podporu. Snahou bylo, aby validní C kód byl validní i v C+, uživatel sám si pak rozhoduje, nakolik a jaké přidané vlastnosti využije. Toto lze dát do kontrastu s multiparadigmovým C++, kde je programátor někdy okolím tlačen do objektového návrhu normami softwarového inženýrství, přestože na daný typ aplikace nemusí být optimální.

Jak ukazuje příklad 3.1, efektivní realizace mnohých vlastností je jednoduchá a při představě odpovídajícího kódu v jazyce symbolických instrukcí i optimální, programátoři jsou však od manuálního psaní podobných nestrukturovaných konstrukcí zrazováni kvůli horší čitelnosti a z toho plynoucí horší udržovatelnosti[1]. Vzhledem k tomu, že významnou část času a peněz ve vývoji SW produktu zabírá údržba kódu[12], je tento aspekt klíčový. C+ tak lze chápat jako mezivrstvu, jako jakýsi rozšířený preprocesor C.

To také naznačuje, jaké typy rozšíření jsou pro C+ vhodné. Např. přetěžování funkcí a operátorů a inicializační funkce (v OOP konstruktor) z C++ jsou snadno implementovatelné, neboť datové typy všech výrazů jsou v době překladu známé a jde tak jen o statický výběr jedné z několika funkcí, naopak nevhodný je systém šablon (natolik komplikovaný, že je sám Turingovsky úplný[2]) nebo polymorfismus (přidaná režie a netriviální implementace ⇒ programátor ztrácí přesnou představu, jak daný kód vnitřně funguje a nepíše pak zcela optimálně).

C+ se snaží zkrátit kód a zvýšit čitelnost, ale *nesnaží* se zvýšit úroveň abstrakce, naopak

[1] Zasloužila se o to známá Dijkstrova stať *Go To Statement Considered Harmful* komentující v té době alarmující problém nadužívaní nestrukturovaných skoků[7].

[2] Což mj. kvůli nerozhodnutelnosti problému zastavení[10] znamená, že nelze určit, je-li libovolný C++ program přeložitelný.

těží např. z toho, že všechny datové typy jsou tzv. *plain old data*. Příkladem je C++ kód `new char[10];`. Nejen, že alokuje 10 bytů, ale implementačně definovaným způsobem někde uloží i informaci o velikosti pole pro dealokaci. Taková metadata jsou např. při předání do modulu v jiném jazyce nedostupná, často je jejich uložení nadbytečné a znemožňují triviální (nízkoúrovňové – `memcpy`, `realloc`) zvětšení pole. Ačkoli takové přetěžování operátorů v C+ lze také chápat jako nástroj abstrakce, určeno je spíše pro účely, kde jsou funkce nahrazující operátory nevyhnutelné, ekvivalentní, pouze méně čitelné (např. dynamické řetězce).

Všechna rozšíření jsou tak popsána i s kanonickou (ideálně nejoptimálnější) implementací v C, programátor tak v každou chvíli ví, jak vypadá výsledný kód a nemusí se spoléhat na optimalizační schopnosti překladače. Ilustrují to (L-hodnotové) reference – v C++ jsou definovány sémanticky, tedy vytvořením alternativní možnosti adresace dané hodnoty, zatímco v C+ jsou záměrně omezeny na implicitně dereferencované ukazatele – uživatel přesně ví, co může výkonostně čekat, je možná spolupráce s čistým C kódem a hlavní přínos (funkce vracející L-hodnoty, nikoli nahrazení ukazatelů) je naplněn.

Nekompatibilita s C je primárně v nových klíčových slovech, snaha je však tento efekt minimalizovat a používat možnosti, co už C nabízí (viz např realizace stukturované vícenásobné selekce v 5.2). Prakticky jde v současné verzi tedy jen o nový operátor `in`, testující výskyt levého operandu v hodnotě napravo. Tento operátor má smysl u abstraktních datových typů jako je seznam či řetězec a jako jiné operátory lze pro vlastní typy přetěžovat, pro zabudované typy podporován není.

Filosofie C+ vede však také k určitým problémům, vycházejícím z modularity jazyka a neznalosti cílové platformy. Nelze např. určit, jestli je některá funkce přetížená nebo ne – vždy může existovat odlišné přetížení v jiném modulu. Výsledkem je, že každá funkce musí být preventivně chápána jako přetížená a jméno každého přetížení unikátně zakódováno. Pro spolupráci s C kódem (zejména knihovnami) je však nutná i podpora nezakódovaných jmen, což musí být řešeno vyjímkou (dále řešeno v 5.5).

Mnoho klíčových faktů je ponecháno na implementaci, jako je znaménkovost typu `char`, přesné rozsahy číselných typů či podstata typů `size_t`,`wchar_t` a `ptrdiff_t`, i přesto, že jsou výsledky zabudovaných operátorů. Tyto informace nejsou v době překladu na C kód jasné a přesto mohou ovlivnit minimálně přetěžování. Co je jestě závažnější, jednotlivé překladače C mohou mít platformě specifická makra, které mohou kondicionálním překladem zcela změnit kód. Pro řešení nejasností musí být překladač v těchto aspektech nakonfigurovatelný[3].

[3]Implementace preprocesing C neřešila, tyto problémy jsou tedy prozatím teoretické.

```
maincycle: for (int i=0;i<argc;i++)
{
  DECL: bool condition=true;
  while (condition)
  {
    switch (argtype(argv[i]))
    {
      case 1:
        if not (i%2)
          break maincycle;
        else
          continue maincycle;
      case 0:
        function1(&condition);
      break;
    }
    else
    {
      function2();
      condition=false;
    }
    break maincycle;
  }
  break;
}
else
  return 1;
return 0;
```

$\Rightarrow$

```
maincycle: for (int i=0;i<argc;i++)
{
  DECL:; bool condition=true;
  while (condition)
  {
    switch (argtype(argv[i]))
    {
      case 1:
        if (not(i%2))
          goto FOR_BREAK;
        else
          goto FOR_CONTINUE;
      case 0:
        function1(&condition);
      break;
      default:
        function2();
        condition=false;
    }
    goto FOR_BREAK;
  }
  break;
  FOR_CONTINUE:;
}
return 1;
FOR_BREAK:
return 0;
```

Obrázek 3.1: Ukázka snadné kanonické konverze některých rozšíření C+

Kapitola 4

Teorie jazyků a překladu

Tato kapitola vysvětluje teoretické základy překladačů a informace o gramatikách formálních jazyků a jejich konstrukci. Dále je konkrétně představena gramatika C+ a její tvorba. Základ terminologie je čerpán z [11], lexikální analýza dále čerpá z [5], syntaktická analýza z [15], speciálně rekurzivní sestup z [9]. Některé informace o tabulce symbolů pochází z [6].

4.1 Formální jazyky a gramatiky

Termín *jazyk* označuje množinu řetězců (vět jazyka), které jsou v jazyce validní. *Řetězce* jsou posloupností (i nulovou) jednotlivých *symbolů*, množina všech symbolů jazyka se nazývá *abeceda*. Pro popis, které řetězce do jazyka patří, se využívá tzv. *gramatika*. Ta je množinou pravidel ve tvaru $A \rightarrow B$, kde A a B jsou jeden nebo více objektů ze skupiny terminálů (symbolů a prázdného řetezce (značen ϵ)) a neterminálů – A však musí obsahovat alespoň jeden neterminál. Tyto pravidla definují povolené přechody, kde je A zaměněno za B. Pro každou variantu posloupnosti přechodů z výchozího stavu (jediného, počátečního neterminálu), která končí zaměněním všech neterminálů za terminály vznikne efektivně řetězec symbolů, patřící do daného jazyka.

Jazyky lze podle složitosti gramatik a výrazové síly rozdělit do čtyř tříd[1]. Každá třída má model, který jazyka dané třídy přijímá – tedy mechanismus, který pro libovolný řetězec určí, zda do jazyka patří či ne. V případě nejjednoduších regulárních jazyků jde o *konečný automat*, což je matematicky pětice stavů, vstupní abecedy, pravidel ve tvaru $stav_A \cdot symbol(nebo\ \epsilon) \rightarrow stav_B$, počátečního stavu a množiny stavů koncových. Lze si ho představit jako graf, kde jsou stavy uzly a jednotlivé hrany odpovídají přechodům s daným symbolem. Pokud začneme v počátečním stavu a každý přechod z řetezce ubere shodný symbol, tak buď zpracujeme celý řetězec a skončíme v koncovém stavu, nebo řetězec v jazyku není. Reálné programovací jazyky jsou většinou definovány vyšší třídou – *bezkontextovými gramatikami*, pro jejíž přijímání je potřeba tzv. *zásobníkový automat*. Jde o rozšíření konečného automatu o zásobník s vlastní abecedou, jednotlivá pravidla jsou rozšířena do formátu $obsah\ zásobníku_A \cdot stav_A \cdot symbol(nebo\ \epsilon) -> obsah\ zásobníku_B \cdot stav_B$. Rozdíl v chování je ten, že aby byl přechod úspěšný, musí být na vrcholku zásobníku 0 až n symbolů označených zde jako $obsah\ zásobníku_A$, ty jsou po přechodu zaměněny na 0 až n symbolů označených jako $obsah\ zásobníku_B$.

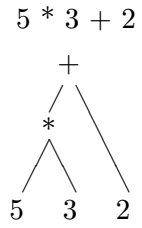
[1] podle Chompského hierarchie[10]

4.2 Bezkontextové gramatiky

Bezkontextové gramatiky mají pravidla ve tvaru $A \rightarrow B$, kde A je právě jeden neterminál a B je jeden nebo více terminálů nebo neterminálů. Cílem *syntaktické analýzy* je vytvořit derivační strom, tedy popsat jaké pravidla a na které neterminály jsou aplikovány (*derivace*), aby vznikl řetězec odpovídající vstupu. V praxi je vstupem program v daném jazyce, který je řetezcem symbolů jazyka. Vytvořením stromu je nejen potvrzena validita vstupu, ale v rámci pravidel může být provedena požadovaná akce – překlad do jiného jazyka nebo např. jako v příkladu 4.1 vyhodnocení.

```
VÝRAZ -> VÝRAZ '+' KRÁTVÝRAZ
VÝRAZ -> KRÁTVÝRAZ
KRÁTVÝRAZ -> KRÁTVÝRAZ '*' ATOM
KRÁTVÝRAZ -> ATOM
ATOM -> <id>
ATOM -> <celé číslo>
ATOM -> <desetinné číslo>
```

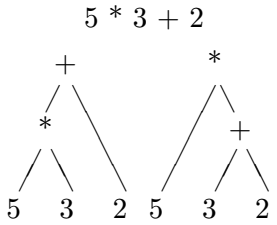
pozn. neterminály jsou velkými písmeny, vše ostatní jsou terminály



Obrázek 4.1: Příklad bezkontextové (LR) gramatiky, popisující výrazy

U špatně navržené gramatiky může nastat, že daná gramatika není jednoznačná, jde s ní pak vytvořit více než jeden validní derivační strom. Kromě toho, i u jednoznačné gramatiky je vhodné specifikovat, na který neterminál bude aplikován další přechod – typicky první nebo poslední, jde pak o nejlevější, resp. nejpravější derivaci.

```
VÝRAZ -> VÝRAZ '+' VÝRAZ
VÝRAZ -> VÝRAZ '*' VÝRAZ
VÝRAZ -> <číslo>
```



Obrázek 4.2: Příklad nejednoznačné gramatiky

4.3 Lexikální analýza

Jak je vidět na příkladech, terminály v gramatice mohou odpovídat i potenciálně mnohoznakovým konceptům, jako je číslo. Reálný překladač má však na vstupu program pouze ve formě posloupnosti znaků. Toto je v praxi zpravidla řešené tak, že se kromě syntaktické analýzy provádí i analýza *lexikální*. Tato druhá úroveň analýzy čte znaky na vstupu a analyzuje je (např. regulární gramatikou pomocí konečného automatu), jejím výstupem je pak posloupnost symbolů (= lexikálních jednotek (lexémů), tokenů) primárního jazyka pro syntaktickou analýzu. Lexikální analyzátor tedy umožnuje mít abstraktnější a jednodušší gramatiku. Mimo to, lex. analyzátor odfiltrovává tzv. bílé znaky, což jsou znaky, které v daném jazyce nemají význam, pouze oddělují tokeny a slouží k formátování zdrojového kódu – běžně jde o mezery, tabulátory a znak nového řádku[2].

- Lexém `<celé číslo>` je tvořen nenulovou posloupností znaků 0-9.
- Lexém `<desetinné číslo>` je tvořen nenulovou posloupností znaků 0-9 následované znakem . následovaný nenulovou posloupností znaků 0-9.
- Lexémy '+' a '*' jsou tvořeny daným znakem.

```
12 + 0.5*
2
```

$$\Rightarrow$$

`␣ 1 ␣ 2 ␣  ␣ + ␣  ␣ 0 ␣ . ␣ 5 ␣ * ␣ nový řádek ␣ 2 ␣`

$$\Rightarrow$$

`<celé číslo>(12), '+', <desetinné číslo>(0.5), '*', <celé číslo>(2)`

Obrázek 4.3: Ukázka lexikální analýzy

4.4 Schéma překladače

Celkové logické schéma při syntaxí řízený překladu je tedy následující. Jádrem je syntaktický analyzátor, který volá lexikální analyzátor, dodávající mu jednotlivé tokeny vytvořené z proudu znaků na vstupu. Syntaktický analyzátor vytváří derivační strom předávaný k sémantické analýze[3] (např. typové kontroly, testy rozsahu platnosti proměnné). Výsledkem je abstraktně syntaktický strom. Typicky pak s ním následuje generování vnitřní reprezentace (např. trojadresným kódem) na kterém jsou provedeny optimalizace a pak převod do cílového jazyka s výstupem.

Poznámka. V tomto projektu bylo mj. kvůli podobnosti vstupu a výstupu výhodnější schéma pozměnit. Vnitřní reprezentací je strom, odpovídající vysokoúrovňové reprezentaci programu, kde jednotlivé uzly rámcově odpovídají jednotlivým příkazům C+. To je výhodné pro snadnou transformaci C+ rozšíření, jejíž kód je pak téměř srovnatelný s popisem v pseudokódu, což je velmi flexibilní a efektivní pro budoucí rozšíření.

[2]Sem lze řadit i komentáře a preprocesorové direktivy, v C jsou však preprocesorem odstraněny před vlastní analýzou.

[3]Syntaktická a sémantická analýza je v praxi často do různé míry provázána a prodáděna paralelně.

4.5 Tabulka symbolů

Důležitá při překladu je i tzv. *tabulka symbolů*, jde o způsob uložení informací nutných při prekladu, zejména identifikátory/proměnné, někdy i klíčová slova či literály. Konkrétní typ obsahu položek a počet tabulek závisí na implementaci, typicky se ale využívají při sémantické analýze např. na zjistění, zda v daném kontextu proměnná už existuje. Pracuje se s ní ve všech částech překladu, vkládá se do ní buď pří lexikální nebo (pokud je LA záměrně odstíněn) až při syntaktické analýze. Tabulka může být globální, pak musí být rozsah platnosti zakódován u jednotivých položek, alternativně je použito více tabulek pro každý rozsah, často pro optimalizaci poskládaných jako jednotlivé rámce na zásobník. Implementovat se dá polem, seznamem, binárním stromem nebo hashovací tabulkou. Jednotlivé varianty mají různé přednosti, důležitou vlastností je zejména rychlost vkládání a vyhledávání.

4.6 LL a LR analýza, rekurzivní sestup

Syntaktickou analýzu lze řesit dvěma způsoby – shora dolů nebo zdola nahorů. Těmto principům odpovídají respektivně *LL* a *LR analyzátory*. LL znamená čtení zleva doprava, s nejlevější derivací (LR s nejpravější).

LL analyzátor využívá LL gramatiku, což je speciální podmnožina bezkontextových gramatik, kde pro každý neterminál platí, že pro libovolný terminál na vstupu lze zvolit nejvýše jedno pravidlo. Toto platí konkrétně pro skupinu LL(1), pro LL(k) musí být zvolené pravidlo rozhodnutelné podle k následujících terminálů. Pro rozhodování, které pravidlo vybrat, se pak využívá tzv. *LL tabulka*. Pro LL(1) je to dvourozměrná matice, pokrývající kombinace terminálů a neterminálů – jednotlivé políčko pak říká, jakým pravidlem rozgenerovat daný neterminál s daným terminálem na vstupu, prázdné políčko znamená chybějící pravidlo, tedy chybu.

```
VÝRAZ -> KRÁTVÝRAZ VÝRAZ-ZBYTEK
VÝRAZ-ZBYTEK -> '+' KRÁTVÝRAZ
VÝRAZ-ZBYTEK -> e
KRÁTVÝRAZ -> ATOM KRÁTVÝRAZ-ZBYTEK
KRÁTVÝRAZ-ZBYTEK -> '*' KRÁTVÝRAZ
KRÁTVÝRAZ-ZBYTEK -> e
ATOM -> <id>
ATOM -> <celé číslo>
ATOM -> <desetinné číslo>
```

Obrázek 4.4: Příklad LL gramatiky

Nejjednoduěí implementace LL analyzátoru je *rekurzivní sestup*. Kromě snadné implementace také nepotřebuje zásobník, informace o neřešených neterminálech jsou uloženy nepřímo v rekurzivních voláních funkcí. Nevýhodou je delší kód, ten však na druhou stranu umožnuje efektivnější chybová a diagnostická hlášení[4]. Pro každý neterminál existuje speciální funkce, ve funkci jsou bloky patřící jednotlivým pravidlům, vybrané podle úvodního terminálu. Uvnitř bloku je pravidlo reprezentováno tak, že neterminály analyzuje rekurzivním voláním odpovídající funkce a terminály pouze přímo porovná se vstupem.

[4]Překladač GCC částečně i kvůli tomuto důvodu přešel od analyzátoru generovaného Bisonem k ručně napsanému rekurzivnímu sestupu[3].

LR gramatiky jsou silnější, tedy dokáží popsat větší množinu jazyků a s typicky méně pravidly (jak je vidět na porovnání příkladů 4.1 a 4.4), LR syntaktický analyzátor s LR tabulkou je však mnohem složitější.

4.7 Konverze LR do LL

Úkolem bylo pro jazyk C+ zkonstruovat LL(1) gramatiku s tím, že bude vycházet z gramatiky C. Oficiální gramatika C je však třídy LR a žádný kompletní překlad do LL nalezen nebyl, ukázalo se tak, že jádrem problému bude převod samotné C gramatiky (která má kolem 70 neterminálů a 80 terminálů) a zda vůbec taková gramatika lze sestrojit.

Předně, je důležité si uvědomit, že ačkoli pro C existuje bezkontextová gramatika, tato LR gramatika *nespecifikuje* reálný jazyk C, pouze jeho nadmnožinu jednodušší pro syntaktickou analýzu. Rozpoznávání validních programů je pak řešeno sémantickými omezeními – např. že ze skupiny specifikátorů třídy uložení může být použit nejvýše jeden nebo že v deklaracích s datovým typem pole může být první rozměr neuveden.

Kromě toho má gramatika vnitřní nejasnosti, první je problém s konstrukcí `typedef`, resp s následným použitím daného synonyma. Když máme např konstrukci `A * B;` může jít o deklaraci proměnné B typu ukazatel na A nebo o výraz s násobením, tedy dva kompletně odlišné významy a derivační stromy. Jádrem problému je, že typové synonymum je stejný terminál jako konvenční identifikátor – problém lze snadno vyřešit trikem tak, že LA vyhledá význam identifikátoru v tabulce symbolů a typ bude chápat jako unikátní terminál.

Druhým problémem je tzv. problém visící else větve, kdy podle gramatiky není v případě více zanořených ifů jasné, ke kterému z nich else patří. Toto je řešeno speciálním pravidlem, že se else vždy přimyká k nejbližšímu ifu. Při LL analýze se tato nejasnost vyřeší implicitně.

Při překladu z LR na LL je potřeba provést dvě věci. První je faktorizace – nesmí existovat dvě a více různých pravidlel začínajících stejně, taková pravidla jsou spojena do jednoho se společnou počáteční částí končící novým neterminálem, s pravidly odpovídající rozdílným částem pravidel původních. Dále je nutné odstranit levou rekurzi, obojí ilustruje příklad.

```
A -> '+' B
A -> '+' C
```

$$\Rightarrow$$

```
A -> '+' A-ZBYTEK
AZBYTEK -> B
AZBYTEK -> C
```

```
A-> A '+' B
A-> <číslo>
```

$$\Rightarrow$$

```
A -> <číslo> A-ZBYTEK
A-ZBYTEK -> '+' B A-ZBYTEK
A-ZBYTEK -> e
```

Obrázek 4.5: Dva typy konverzí při překladu z LR do LL

Kromě LL gramatiky je potřeba zkontruovat i LL tabulku, její uspěšné vytvoření také dokáže jednoznačnost gramatiky. Pro to je nutné zkonstruovat pro kazdý neterminál následující:

- Množina *first*, která obsahuje v pravidlech bezprostředně následující terminály.
- Příznak *empty*, pokud některé pravidlo může být rozderivováno na ϵ.
- Množina *follow*, která obsahuje všechny terminály, které se mohou jakkoli vyskytovat napravo od daného neterminálu.

Následně jde pro kazdé pravidlo určit terminály, které na něj vedou – pokud začíná terminálem, je očividně vybráno při daném terminálu. Pokud začíná neterminálem, je vybráno, pokud začíná terminálem z množiny *first* daného neterminálu, pokud je neterminál *empty* pak také z množiny *follow*.

4.8 Gramatika C+

Gramatika C+ se snaží z praktických důvodů držet mateřské LR gramatiky C. Je složená ze čtyř hlavních částí.

Výrazy jsou řešeny jako v příkladu 4.1 a překlad do LL je jednoduchý, až na to, že kulaté závorky s podvýrazem musí být řešeny na stejné úrovni jako přetypování.

Příkazy zahrnují všechny řídící konstrukce a objevuje se zde problém – v případě návěstí nelze jednoznačně určit pravidlo podle prvního tokenu, je nutný náhled o dva. Toto však jde vyřešit postupem znázorněným v příkladu 4.6 tak, že na daném místě provedeme rozgenerování neterminálů natolik, aby šlo sjednotit obě konfliktní pravidla a vyřešit nejasnost druhým tokenem – nesmíme však zapomenout dodat sekvenci, která rozgenerováním vznikla.

```
PŘÍKAZ -> VÝRAZ
PŘÍKAZ -> <id> ':' PŘÍKAZ
```

$$\Rightarrow$$

```
PŘÍKAZ -> <id> NÁVĚSTÍ-NEBO-VÝRAZ
PŘÍKAZ -> <celé číslo> SEKVENCE-VÝRAZ-NA-ATOM
PŘÍKAZ -> <desetinné číslo> SEKVENCE-VÝRAZ-NA-ATOM
NÁVĚSTÍ-NEBO-VÝRAZ -> ':' VÝRAZ
NÁVĚSTÍ-NEBO-VÝRAZ -> SEKVENCE-VÝRAZ-NA-ATOM
SEKVENCE-VÝRAZ-NA-ATOM -> KRÁTVÝRAZ-ZBYTEK VÝRAZ-ZBYTEK
```

Obrázek 4.6: Ilustrace problémů při překladu z LR do LL a jejich řešení

Třetí částí jsou **deklarace**, zde se nachází jeden problém s nerozlišitelností – neterminály `deklarator` a `abstract-declarator`, popisující deklaraci s identifikátorem a bez (použitou pro samotnou specifikaci typu) se dají libovolně uzávorkovávat, rozlišit mezi nimi je tak velmi těžké. Oboje zároveň mohou být pouze v argumentech funkcí, byl tu tedy vytvořen nový neterminál kombinující oboje. Rozlišení je pak otestováním, zda nejvnitřnejší závorka obsahuje identifikátor nebo ne.

Poslední částí je **nejvyšší úroveň**, zde jsou v LR pravidla pro běznou deklaraci a pro definici funkcí, v LL jsou však az na závěr nerozeznatelné, proto existuje nový neterminál

extérní deklarace, která navíc (oproti deklaraci) v případě funkcí může být následována definicí.

Rozšiřující části C+ se v gramatice nijak významně neprojevují, buď imitují stávající funkcionalitu (jako else větve u cyklů nebo operátor `in`) nebo jsou řešeny zcela mimo gramatiku (viz odsazování bílými znaky v kapitole 5.1 a separátory v kapitole 5.6). Vzhledem k velkému rozsahu výsledné gramatiky (přes 200 neterminálů) byl napsán speciální program, automatizující tvorbu LL tabulky.

Kapitola 5

Rozšíření v jazyku C+

5.1 Odsazování bílými znaky

Jednou z rozšiřujících možností zápisu, vedoucímu ke kratšímu kódu, je tzv. odsazování bílými znaky, v originále *whitespace indentation*. Tato technika se původně objevila v jazyce Python.

Jedná se o odlišnou filosofii psaní kódu, kde nejsou jednotlivé podbloky ohraničeny pomocí konkrétního symbolu (v C složených závorek) v textu, ale vychází z odsazení daného řádku – řádek s delším odsazením je považován za podblok.

Podobný systém není v syntaxi programovacích jazyků obvyklý a mezi programátory má jak zastánce, tak odpůrce. Klíčovým pozitivem je zkrácení kódu, kdy většina odsazovacích stylů sází koncovou závorku a mnoho, zejména Allmanův a styl doporučovaný pro GNU, i počáteční závorku na nový řádek. U zkušenějších programátorů se dá předpokládat dodržování konzistentní velikosti odsazení už kvůli čitelnosti, jeho striktní vyžadování pak může být z hlediska vyšší čitelnosti bráno jako výhoda. Naopak odpůrci zmiňují následující fakta: zatímco v konvenčních jazycích lze zdrojový kód chápat jako proud tokenů, zpracování podbloků zde vyžaduje speciální logiku, což komplikuje syntaktickou analýzu. Při mnohořádkových blocích, jaké mohou vznikat například v těle konstrukce `switch`, může být problém po dlouhé vertikální mezeře určit správné odsazení, kdy špatná volba vede k validnímu, ale zcela odlišnému kódu. Nekonzistence v typu odsazení (tabulátory oproti mezerám) může být z pozice estetického problému s čitelností povýšena na problém syntaktický. Tabulátor totiž nemá vůči mezerám pevně definovanou velikost, ta se typicky pohybuje mezi hodnotami 1 až 8. Kombinace obojího pak při změně poměru může vést k nesprávnému odsazení. Kromě toho, mnoho textových editorů, vývojových prostředí a webových stránek do různé míry provádí automatické formátování zdrojových textů, což může opět způsobit nečekané chyby.

V návrhu jazyka byly tyto fakty zohledněny a výsledkem je hybridní systém, kde je odsazování bílými znaky zcela volitelné a může být kombinováno i s běžnými bloky podle potřeb programátora. Řídící konstrukce (`if`, `while`, `for`) a definice funkcí mohou být po kulatých závorkách následovány dvojtečkou, která uvozuje speciální blok. Ten začíná hned za dvojtečkou a zahrnuje i všechny následující řádky, dokud je jejich počáteční odsazení (tedy do prvního tokenu na řádku) vyšší než na řádku, obsahující dvojtečku. Stejný systém funguje nezávisle pro else větve cyklů. Takovýto blok je funkčně ekvivalentní běžnému podbloku ve složených závorkách. Je možné a kompaktní, aby takovýto blok byl pouze na jediném řádku s počáteční dvojtečkou, nesmí však být prázdný – umožnění tohoto by vedlo k těžko objevitelným chybám. Dvojtečka byla pro tuto konstrukci zvolena jednak kvůli

dobé čitelnosti, dále pak pro návaznost na Python, který ji používá jako ukončovací znak podmínek i při definici těl funkcí.

```
int fakt(int x):
  if (x == 1):
    print(x);
    return 1;
  else: print(x);return x*fakt(x-1);
```

$\Rightarrow$

```
int fakt(int x)
{
  if (x == 1)
  {
    print(x);
    return 1;
  }
  else
  {
    print(x);
    return x*fakt(x-1);
  }
}
```

Obrázek 5.1: Ukázka odsazování bílými znaky

Samotná detekce konce speciálního bloku není řešena v gramatice, překladač si tak musí při překladu každého řádku uchovávat jeho počáteční odsazení. Rozeznání nového řádku je snadné, neboť nový řádek je v rámci lexikální analýzy brán jako unikátní druh tokenu (dále viz 5.6). Odsazení následujícího řádku je pak vždy analyzováno a pokud je menší, než odsazení posledního analyzovaného bloku odsazeného bílými znaky, je vygenerován speciální token značící konec bloku (potenciálně opakovaně).

Tabulátor je chápán jako dvě mezery, při kombinaci různých bílých znaků je však programátor varován.

Poznámka. Definice bílými znaky odsazeného bloku se drobně liší od Pythonu, což je vidět v následujícím příkladě. Zatímco v Pythonu je takováto else větev považována za nejasnou a tedy nelegální, zde patří k prvnímu `if`u. Nutno dodat, že podobné vnořování není kvůli čitelnosti doporučeno a lze přirovnat k problému nejasné else větve v konvenční gramatice zmíněnému výše.

```
if (x): if (y): print(y);
else: print("X IS FALSE");
dalsi_funkce();
```

$\Rightarrow$

```
if (x) { if (y) { print(y) } }
else { print("X IS FALSE"); }
dalsi_funkce();
```

5.2 Rozšíření řídících konstrukcí

Jazyk C+ obsahuje řadu rozšíření řídících konstrukcí, umožňující přehlednější a abstraktnější kód.

Na konceptuální úrovni je však jediné nutné řízení programového toku je možnost kondicionálního a nekondicionálního skoku. V jazyce symbolických instrukcí a strojovém kódu, jenž je obvykle výsledkem kompilace jazyka C, jsou tyto instrukce určeny k realizaci všech řídících struktur - libovolné iterace a selekce. Lze tedy říci, že pomocí C ekvivalentů těchto skoků, tedy příkazu `goto` a `if`, je možné nahradit libovolnou, abstraktnější řídící konstrukci, která je do jazyka symbolických instrukcí převoditelná.

Není však praktické tímto způsobem psát kód. Jazyk C je jazykem strukturovaným a těží tak z jisté míry abstrakce. Ačkoli je pro potřeby optimalizace velmi příjemné, že si lze za řádky kódu přímo přestavit odpovídající posloupnost instrukcí, použití vhodnějších bloků, jako je cyklus `for`, lépe vystihuje význam kódu, což společně s kratším zápisem redukuje potenciální chyby, dává lepší šanci na automatickou optimalizaci a zlepšuje čitelnost, což pomáhá při údržbě.

Prvním vylepšením je možnost použití else větví mimo `if` i u cyklů, tedy u konstrukcí `for`, `while` a `do-while`. Tato technika je opět převzata z Pythonu. Sémanticky se else větve chovají stejně jako u `if`u, jejich kód je proveden, pokud je podmínka daného cyklu vyhodnocená jako nepravdivá. Vzhledem k tomu, že kód za cyklem je běžně prováděn vždy po ukončení cyklu, je na místě otázka, k čemu je explicitní else větev vhodná. Rozdílem je chování příkazu `break`, který obsah else větve efektivně přeskočí. Tento princip je velmi vhodný zejména pro častou operaci vyhledávání iterací nad datovou strukturou cyklem `for`, kde jde jasně rozeznat případ nalezení (kód končící `break`) od ukončení průchodu v else větvi.

Pro konzistenci s ostatními strukturami byla přidána možnost else větve i pro konstrukci `switch`, kde tato větěv přestavuje alternativní zápis k neošetřené variantě. Z toho plyne, že nelze použít zároveň s návěstím `default`. Jde čistě o strukturovanější syntaktickou změnu, v rámci implementace je tato větev přemístěna do těla `switch`e za přidané návěstí `default`. Drobným rozdílem, vycházejícím z gramatiky, je to, že deklarace v těle `switch`e nejsou v else větvi viditelné. Takové deklarace (s případnou inicializací) jsou však v toku kódu stejně přeskočeny, což v C není doporučeno a ve striktnějším C++ dokonce zakázáno.

Poznámka. Dodání nových else větví do gramatiky může způsobit neočekávanou nekompatibilitu s C, např u následující konstrukce, kde se `else` přimyká k cyklu `while` namísto k `if`u. Problém lze odstanit explicitním uzávorkováním či použití odsazení bílými znaky.

```
if (x)
  while (y) {}
else {}
```

Pro konstrukci `switch` bylo navrženo i další rozšíření. Obsah jeho těla totiž odpovídá nestrukturovanému kódu, návěstí `case` a `default` mohou být umístěna prakticky libovolně, což s úspěchem využívá např. takzvaný *Duff's device*, vysoce optimalizovaný algoritmus rozkládající cykly při sériovém kopírování. Bohužel pro typičtější použití, kdy je `switch` použit pro několikanásobnou selekci podobnou opakované posloupnosti `if - else if` s jediným vyhodnocením podmínky, je standardní chování zbytečně komplikované a naopak vede u méně zkušených programátorum k chybám jako zapomenutý `break`, kdy se neočekávaně provádí i další varianta. I bez chyb je však psaní kódu s mnoha variantami, kde má každá formát `case <hodnota>: příkaz; break;` příliš zdlouhavé.

Namísto navržení nové, strukturovanější varianty `switch`e, což by si vynutilo přidání nového klíčového slova, byla realizována flexibilnější varianta – alternativní syntax `case`, které se nechová jako návěstí, ale jako řídící konstrukce, podobně jako např. `if`. Z realizačního hlediska je jeho obsah roven běžnému case následovaným tělem, pouze je za konec těla automaticky dodán `break`. Rozlišení v syntaxi je provedeno klíčovým slovem `do` za podmínkou (viz příklad 5.2).

Jednou z největších limitací C syntaxe je nemožnost jednoduchého ukončení zanořených cyklů. Ačkoli lze situaci vyřešit velmi snadno použitím `goto`, tato možnost je často odsuzována a vede k použití pomocných proměnných. Ty jsou často nejen méně efektivní, ale přidají dost pomocného kódu na to, aby i čitelnost byla horší než se zatracovaným `goto`.

Řešení v C+ je pak vhodný kompromis – z pohledu uživatele lze před cyklus přidat návěstí a následně pomocí `continue` či `break` s daným návěstím provést daný příkaz, z pohledu implementace jde o příkaz `goto` skákající buď na konec cyklu u `continue` či za konec cyklu (a případné else větve) u `break`u. Nejen, že je `goto` od uživatele odstíněno, není také jednoduché udělat chybu jako při ruční realizaci a navíc je řešení tohoto problému v celém programu jednotné.

Dalším malým vylepšením je, že návěstí mohou stát samostatně jako jiné příkazy. V C gramatice totiž návěstí uvozuje příkaz, což v praxi znamená, že nesmí stát před deklarací ani na konci bloku. Toto omezení nemá význam a zejména pro návěstí uprostřed deklarací s inicializacemi nebo návěstí na konec cyklu velmi překáží. Řešení je triviální, za takové návěstí přidat prázdný příkaz. Dá se čekat, že každý běžný překladač takto dokáže vygenerovat optimální kód – GNU C dokonce stejné rozšíření podporuje implicitně.

```
number number__add(number a,number b)
{
  number tmp;
  switch (a.type):
    case NUM_INT do:
      switch (b.type):
        case NUM_INT do:
          tmp.type=NUM_INT;
          tmp.value.i=a.value.i+b.value.i;
        case NUM_FLOAT do:
          tmp.type=NUM_FLOAT;
          tmp.value.i=a.value.i+b.value.f;
      else: assert(0);
    case NUM_FLOAT do:
      switch (b.type):
        case NUM_INT do:
          tmp.type=NUM_FLOAT;
          tmp.value.f=a.value.f+b.value.i;
        case NUM_FLOAT do:
          tmp.type=NUM_FLOAT;
          tmp.value.i=a.value.f+b.value.f;
      else: assert(0);
  else: assert(0);
  return tmp;
}
```

pozn. používá datový typ `number` definovaný v příkladu 5.4

Obrázek 5.2: Využití strukturované konstrukce `switch`

5.3 Implicitní argumenty

Je umožněno, podobně jako v řadě vyšších jazyků, v deklaraci funkce argumentům přiřadit implicitní konstantní hodnoty. Pokud je pak taková funkce zavolána, argumenty po posledním povinném (tedy argumentu bez implicitní hodnoty) nemusí být specifikovány.

Překlad do C je jednoduchý, chybějící parametry jsou do volání doplněny. Tento přístup však očividně nevede k nejrychlejšímu možnému výsledku, přímý překladač do jazyka symbolických instrukcí jako má C++ si může dovolit speciální optimalizace. Jednou z optimalizačních možností, jak zabránit zbytečným argumentům při volání, je vytvořit další přetížení funkce s duplicitním kódem, ale menším počtem argumentů. Cenou je pochopitelně zvětšení programu.

```
/**
  Rekurzivni funkce, ktera projde stromem a podle hodnoty druheho argumentu
   provadi zmeny. U korene (v prvni iteraci) je vzdy hodnota true.
**/
bool tree_set_void(Tree *tree,bool swap=true);
/**
  Knihovni funkce, ktera vyzaduje prilis argumentu, ktere jsou vsak
  zbytecne. Je efektivnejsi dodat deklaraci s implicitnimi argumenty, nez
  pouzit napr. adapter, tedy funkci obalujici volani
**/
Stream *API_openStream(String filename,Flags arguments,Device_id
  device=DI_HARDDISK,Stream *duplicate=NULL);
```

Obrázek 5.3: Vhodné případy použití implicitních argumentů

5.4 Vlastní datové typy

Jedna z klíčových vlastností C++ je možnost komplexní definice nových datových typů i s funkcemi, které nad nimi operují – metodami. S použitím redefinice operátorů lze v C++ vytvořit datové typy, které se používají stejně snadno jako typy zabudované. Přestože C+ nemá zájem stát se objektově orientovaným jazykem, faktem zůstává, že i v čistém C je běžné vybudovat pro uživatelem definovaný datový typ (např. struktura realizující zásobník či dynamické pole) podobně ucelené funkční rozhraní. Vhodná podpora ze strany jazyka tak opět může pomoci ke zkrácení kódu.

Co se týká definice toho, co je za tzv. vlastní typ považováno, se od C++ liší – tam je vlastním typem chápána jen třída (`class`) nebo struktura (`struct`)[1] s rovnou definovanými metodami.

C+ se snaží být flexibilnější, každé použití konstrukce `typedef` definuje nový vlastní datový typ. To je v přímém protikladu s běžným chováním C i C++, kde `typedef` vytváří pouhé synonymum k základnímu typu. Vzhledem k tomu, že tento vlastní typ se může implicitně konvertovat na základní typ, není tento rozdíl v obvyklém C-kompatibilním kódu patrný. Ten se může projevit jen při dvou příležitostech – při přetěžování funkcí (viz kapitola 5.5) a při předefinování operátorů.

V kontextu jazyka C+ termín metoda označuje funkci se speciálním významem pro konkrétní vlastní datový typ. Po syntaktické stránce se neliší od normální funkce a lze ji

[1] Rozdíl mezi nimi je jen v implicitním nastavení viditelnosti členů.

```
/**
  Priklad dynamickeho obecneho ciselneho typu
  s vhodne pretizenymi aritmetickymi operatory
  se muze chovat stejne jako interni aritmeticke typy
**/
typedef
  struct number
  {
    enum NumberType
    {
      NUM_INT, // Cele cislo
      NUM_FLOAT, // Desetinne cislo
    } type;
    union NumberValue
    {
      long i;
      double f;
    } value;
  }
number;
/**
  Vyctove literaly definujici novy vlastni typ maji prave definovany typ
  - tedy "bool", nikoli "enum enum_bool"
**/
typedef
  enum enum_bool {false,true}
bool;
```

Obrázek 5.4: Vhodné typy pro přetížení operátorů

explicitně volat jako každou jinou, její jméno však má speciální formát `typ__metoda`. Tento způsob je inspirován Pythonem, naopak řešení z C++ (využívající speciální klíčové slovo a daný operátor) se neukázalo praktické, mimo jiné kvůli rozlišení pre- a postinkrementace. První argument funkce musí typem odpovídat. Pokud je dané jméno metody validní a pokud argumenty odpovídají odpovídajícímu operátoru, začne být metoda platná ihned po deklaraci (či definici).

Metody odpovídají jednotlivým operátorům. Odpovídající použití operátoru s danými datovými typy ve výrazu se přemění na volání adekvátní metody. Nelze přetížit operátory čárka, tečka, ternární operátor a operátor volání funkce.

Přetěžování operátorů má však i stinné stránky. Kritici zdůrazňují, že chyby v metodách se velmi špatně odstraňují, neboť programátor si ani nemusí uvědomit, že daný kus kódu metodu volá, tedy kód ztrácí svoji jednoznačnost. Tento problém je v C+ částečně pokryt tím, že výstup v čistém C je pro ladění dostatečně čitelný.

Další kritika se týká specificky C++, a to složitého systému výběru správného přetížení, který může zahrnovat až dvě standardní a jedno uživatelem definované přetypování. Tento systém není dokonalý, což dokumentuje např. *„safe bool“ problém*, kdy objekt, který je konvertovatelný na `bool` (pro použití v podmínkách), je automaticky konvertovatelný i na `int`, což je často sémanticky nepřijatelné a musí být řešeno komplikovanými triky. To je zejména důvod, proč norma C++11 přidala explicitní konverze, na které však lze nahlížet jen jako na záplatu. C+ se v tomto rozhodl držet přístupu z Pythonu – lepší explicitně než implicitně – a povoluje přetížení jen s identickými typy či s typem odpovídající základu vlastního typu.

```
// Vlastni datovy typ vector, ktery je polem tri celych cisel
typedef int vector[3];
// Deklarace metody pro typ vector pro scitani dvou vektoru
vector vector__add(vector a,vector b);
// Deklarace metody pro typ vector pro scitani vektoru s cislem
vector vector__add(vector a,int val);
// Deklarace metody pro typ vector pro nasobeni vektoru s cislem
vector vector__mul(vector a,int val);
/**
  Specialni varianta typu int, ktera explicitne testuje deleni nulou. Je
  dulezite zduraznit, ze pokud napr. operace scitani nebude definovana
  explicitne, pak int_dbg + int_dbg vraci int! (diky implicitni konverzi)
**/
typedef int int_dbg;
int_dbg int_dbg__div(int_dbg a,int_dbg b)
{
  assert(b != 0);
  return a/b;
}
```

Obrázek 5.5: Příklady přetěžování operátorů

5.5 Přetěžování funkcí

Přetěžování funkcí je mechanismus běžný v mnoha vyšších, převážně staticky typovaných jazycích jako C++ a Java. Základní myšlenkou je, že funkce s konktrétním jménem nemusí být pouze jediná, ale může existovat celá množina funkcí se stejným jménem lišící se v počtu a nebo typech přijímaných parametrů, lišící se tzv. *signaturou*. Podle počtu a typu argumentů ve volání funkce je pak vybrána odpovídající funkce s kompatibilní signaturou.

V objektově orientovaném prostředí je tento mechanismus vhodný pro realizaci polymorfismu, kdy jeden typ zprávy zasílaný různým objektům (např. výpis obsahu metodou print) má pro každný typ unikátní implementaci. Stejná funkcionalita je ale výhodná i pro C+, např. funkce abs, která může vracet absolutní hodnotu pro typy celočíselné, s plovoucí desetinnou čárkou i komplexní.

V běžném C jsou podobné případy řešené vytvořením několika funkcí s odlišnými jmény, např. abs, labs a fabs pro typy `int`, `long int` a `double`. I v nemodifikovaném C je však snaha výhody přetěžování částečně emulovat a do jazyka tak byly od verze C11 přidány tzv. typově generická makra, které podle datového typu argumentu vyberou vhodnou variantu.

Jazyk C+ tedy zavádí robustnější implementaci téhož, tedy každé přetížení funkce dostane speciální, unikátní jméno a následně je na místě volání vybráno odpovídající přetížení.

Poznámka. Vzhledem k tomu, že je výběr jednoznačně určitelný již při překladu, nemá systém žádnou režii. Pro srovnání, C++ umožňuje také díky virtuálním funkcím realizovat skutečně dynamický polymorfismus, což však má negativní vliv na výkon (nutnost uchovávání metainformací o typu a průchod tabulky virtuálních metod), který je před programátorem zcela skrytý. Toto odporuje filosofii C+, stejnou funkcionalitu lze však v případě zájmu realizovat manuálně i v čistém C, potenciálně optimalizovaněji pro konkrétní situaci.

Prvním problémem je správný výběr formátu zakódovaného jména funkce. Je jasné, že formát musí obsahovat jméno funkce a specifikaci všech parametrů, možností je však mnoho. Jednou z nevýhod C++ je, že standard ponechává implementacím volnost v re-

alizaci částí jako je obsluha vyjímek, virtuální tabulky či právě formáty jmen funkcí. To způsobuje, že moduly zkompilované různými překladači nejsou kompatibilní. Naopak, rozdíly ve formátech jmen je často záměrné, aby zamezily linkování nekompatibilních modulů.

V C+ (resp. C) naštěstí problémy s kompatibilitou překladačů nejsou, je tedy v tomto směru důležité hlavně to, aby byl formát navržen kompatibiliní s budoucími verzemi jazyka.

Překladač	void func(int,char)
GNU v.2	`func__Fic`
GNU v.3	`_Z4funcic`
Microsoft Visual C++	`?func@@YAXHD@Z`
Watcom C++	`W?func$n(ia)v`

Obrázek 5.6: Porovnání způsobů zakódování jmen mezi překladači[8]

Jak je vidět v tabulce 5.6, překladače často využívají některou formu maďarské notace, výsledky se však radikálně liší. Pro posouzení kvality systému lze použít tyto parametry

1. kompatibilita – formát musí být adekvátně přenositelný, rozšířitelný do budoucna a unikátní pro každou funkční signaturu

2. kolize s obyčejnými jmény – pokud formát nepožívá speciální symboly nepovolené v C identifikátoru, může dojít ke kolizi s nezakódovaným jménem funkce či proměnné. Méně čitelný formát toto riziko snižuje

3. čitelnost – přestože formát je z větší části interní záležitostí, i běžný programátor se s ním někdy může setkat, např. v chybových hlášeních linkeru nebo debuggeru

4. délka – čitelnější formát je zpravidla delší, při příliš dlouhých hrozí naopak ztráta čitelnosti. Kromě toho, překladače C mohou dle standardu omezit maximální délku identifikátoru (resp. počet signifikatních znaků)

Všechny tyto faktory byly při návrhu zváženy. Jazyk C+ si cení inituitivní čitelnosti výsledného kódu i na úkor vyšší délky identifikátorů. Vzhledem k tomu, že výsledkem je konvenční C kód a jména funkcí jsou stále standardní C identifikátory, kolize jsou nevyhnutelné.

Podobnou problematiku standard C řeší identifikátory rezervovanými pro budoucí rozšíření a identifikátory rezervovanými pro implementaci překladače a , ty začínají dvěma podtržítky nebo podtržítkem a velkým písmenem – v tabulce výše si lze všimnout prefixu `_Z` tohoto využívajícího. C+ rezervované identifikátory nevyužívá, kromě potenciálních kolizí s knihovnami a překladačem C i pro nižší čítelnost.

Naopak, návrh C+ formátu věří, že výhody jasně čitelného formátu převáží nad velmi nepravděpodobnou situací, kdy bude omylem nová funkce či proměnná pojmenovaná přesně shodně s zakódovaným jménem funkce jiné. Z hlediska překladače se deklarace funkce chová, jako kdyby zároveň deklarovala i zakódovaný identifikátor, případné kolize či zastínění by tak měl překladač odhalit v rámci sémantické analýzy.

Formát je jednoduchý, začátek je jméno funkce, následované seznamem argumentů. Pokud je argument `void`, je to vyjádřeno jako `_0`, pro proměnlivý počet argumentů je přidané podtržítko na závěr. Konečně, subvarianta `void` (vysvětleno dále) končí `_v`. Každý argument začíná podtržítkem, následovaný jednoznakovými typovými kvalifikátory (c,v,r) a rozlišením, zda jde o `enum` (E),`struct` (S) nebo `union` (U). Následuje jméno typu předcházené

jeho velikostí, což řeší nejasnosti se jmény typů obsahující podtržítko (a navíc zmenší šanci na to, že si programátor podobné jméno funkce vybere spontánně). Na konci jsou popsány části typu mimo jeho základ, tj. ukazatele (p)[2] a funkce (f, následuje počet a podseznam argumentů ukončený podtržítkem navíc), v pořadí od základu k identifikátoru (viz 2.2).

```
extern double strtod_c4charpr_4charppr(const char *restrict, char
  **restrict);
int main_c4charpp_3int(const char *argv[],int argc);
void parse_block_list_6Parserp_v(Parser *parser);
int print_0(void);
int print_E5Color(enum Color);
int print_c4charp_(const char*const,...);
int print_3intlU(unsigned long);
int print_4bool(bool);
```

Obrázek 5.7: Příklady přetížených funkcí s aplikovanými zakódovanými jmény

Samotný princip rezoluce je inspirovaný C++, základem je posouzení celé množiny funkcí a ohodnocení, nakolik daná funkce odpovídá udaným parametrům. Pokud existuje jediná funkce s nejlepším hodnocením, než mají ostatní, je zvolena, jinak je přetížení nejasné.

C+ má však oproti C++ několik klíčových rozdílů, které se promítají do hodnotící funkce. Jednotlivé vlastní typy nejsou identické i při stejném základu (vede na rozdělení přesné shody na 3 úrovně). Pro zpětnou kompatibilitu s C lze provést i konverze v C++ nemožné, mají však velmi nízkou prioritu.

1. přesná shoda
2. vlastní typ na základ
3. vlastní typ na jiný vlastní typ
4. standardní povýšení (na int nebo unsigned int)
5. libovolné povýšení
6. aritmetické konverze (mezi desetinnými, celočíselnými, booleovským i komplexními typy, znaménkové i neznaménkové, degradace na nižší typ)
7. libovolné konverze (mezi ukazateli,výčtovými typy a čísly)
8. na místě s proměnlivým počtem parametrů

Obrázek 5.8: Hodnotící funkce srovnávající konverzi z typu ve volání funkce na typ v daném přetížení

Specialitou C+ je, že umožnuje přetížit nejen podle argumentů funkce, ale i podle návratového typu. Nejde však o tradiční přetížení, jako spíše o optimalizaci. Některé funkce, jako např. knihovní printf, vrací informativní hodnotu, která ve valné většině případů není

[2] V rámci funkční signatury jsou pole identické se stejně kvalifikovanými ukazateli.

využita. Takové funkci pak jde přidat optimálnější subvariantu nevracející žádnou hodnotu – void subvarianta ma identickou signaturu, pouze je volána ve všech kontextech, kde návratová hodnota není čtena.

Přetěžování funkcí však vede i k problémům. Při spojování s moduly, napsanými v čistém C, tedy zejména standardní knihovny, je potřeba nezakódovávat jména funkcí. C++ toto řeší pomocí zvláštního prostředí `extern ''C''`, C+ rozezná čisté C pomocí jiné přípony v direktivě include[3]. C je navíc flexibilnější než C++ a umožnuje i pro funkci z čistého C přidat další přetížení, nicméně z čistého C může být nejvýše jedna varianta.

Funkce s nespecifikovaným počtem parametrů je v C+ i C++ brána jako by neměla žádné, při převzetí z čistého C je však kvůli kompatibilitě volatelná s libovolným počtem.

Další nejednoznačností je rezoluce typu ternárního operátoru za předpokladu, že v obou větvích jsou funkce které mají více subvariant. V takovém případě má vždy prioritu subvarianta vracející hodnotu.

Problémem jsou také operace s funkčními ukazateli. Identifikátor funkce je pro mnoho operátorů implicitně konvertován na ukazatel na funkci a lze s ním provádět srovnatelné operace. Pokud ale značí přetíženou funkci, je jeho datový typ často neurčitelný, proto není tato alternativa povolena.

5.6 Volnější syntaxe

Důležitou částí práce bylo zjistit, zda gramatika C obsahuje nadbytečné části, idenfitikovat je a učinit je volitelnými. Průzkumem jazyků, které mají gramatiku inspirovanou C a mají laxnější syntax, jako je Python nebo GML, byli objeveni dva kandidáti – středníky a kulaté závorky u podmínek v řídících cyklech.

```
typedef
  struct Node
  {
    struct Node *root
    struct Node *left
    struct Node *right
    int value
  }
Node
bool test(Node *node)
int func(Node *node,int x,int y)
{
  if (x >= 3)
    func(node->left,x-1)
  else
    func(node->right,y-1)
  y++
  int result=test(node->root)?x:y
  return result
}
```

Obrázek 5.9: Vzorek kódu s absencí středníků

[3]Podpora prepocessingu však implementována nebyla.

Na příkladu 5.9 je vidět, že při odstranění středníků z obvyklého kódu lze stále jednoznačně určit původní význam. Přesto však existují nejasné situace.

Na nejvyšší úrovni problém není, protože deklarace vždy začínají deklaračními specifikátory, které jsou nezaměnitelné. Potenciální problém s vlastními typy, které jsou technicky běžné C identifikátory, je vyřešen tak, že na úrovni syntaktické analýzy jde o odlišný terminál (viz 4.8). Stejná situace je s deklaracemi v těle funkce, kde je jejich začátek i konec snadno identifikovatelný. Totéž platí i pro většinu příkazů, neboť klíčová slova tvoří přirozené hranice. Jediný, ale zato výrazný problém způsobují dva po sobě následující výrazy, kde absentující středník může snadno ze dvou výrazů udělat jeden.

I kdybychom problém nejednoznačnosti dočasně ignorovali, nevýhodou odstranění středníků zůstává snížení čitelnosti, proto je po vzoru Pythonu možné vynechat středník jen na místě přelomu řádku. Při jiném náhledu na věc můžeme středník označit jako tzv. tvrdý separátor a nový řádek jako separátor měkký, tedy potenciální.

V gramatice je tento princip realizován tak, že na místech terminálu středník je abstraktnější terminál separátor. Konceptuálně znamená, že pro zvolení daného gramatického pravidla na daném místě přijde středník, nebo nový řádek – ten je ale chápán jako *potenciální* středník a překlad probíhá dále, jako by tam nebyl. Pouze pokud při překladu nastane chyba, překlad se navrátí k poslednímu potenciálnímu středníku, ten je nyní považován za skutečný a překlad se od daného místa opakuje. Ve všech ostatních kontextech je měkký separátor zcela ignorován.[4]

Tento systém je sice dostatečný, ale syntaktický analyzátor musí být schopný ukládat kontext překladu, což je zvlášť u rekurzivního sestupu programovým zásobníkem problém. Nicméně, při hlubší analýze gramatiky lze zvolit snažší implementaci. Kromě zmíněné situace se středníkem uprostřed výrazu jde každé pravidlo s přijatým měkkým separátorem rozřešit hned podle prvního tokenu po novém řádku.

U výrazů uvažujme dva výrazy rozdělené měkkým separátorem ve tvaru A `<nový řádek>` B, kde A je ukončený výraz (za A může přijít středník). Jediná možnost, jak může být B validní, samostatný výraz a zároveň může být chápáno jako součást A je, pokud B začíná na unární operátor, který je zároveň operátorem binárním, tj. $\&, *, +, -$. [5] Z těchto operátorů má praktický smysl jen dereference, ovšem s tou typicky spojená operace přiřazení nebude validní, neboť po spojení A a B násobením patrně nebude na levé straně L-hodnota. V úvahu ale přichází postinkrementace, máme tedy konkrétní příklad reálně použitelného kódu, kvůli kterému se při odstranění středníků kód vyhodnocuje odlišně od intuitivního chápání.

```
func(1,2)
*p++

if (x == 3)
  *p--;
```

Obrázek 5.10: Příklady nejasného kódu při uvolnění syntaxe

Druhý případ také ukazuje, proč musí být podmínka explicitně uzávorkována. Pokud by podmínkou byl libovolný výraz, byl by takový kód chápán jako

[4] Na místě separátoru je akceptovatelný i znak }, v každém použití totiž implikuje předchozí středník.
[5] Speciální případ jsou kulaté závorky, pokud výraz A končí identifikátorem funkce

```
if ((x == 3)*p--)
  ;
```

a na rozdíl od dodání středníku v předchozím případě toto není bez složených závorek jednoduše řešitelné.

Do jazyka však byly přidány dvě drobná rozšíření podmínek. Lze je negovat, aniž by bylo nutné psát nadbytečné kulaté závorky a podmínka `while` cyklů navíc může být prázdná, což má chování konzistentní s cyklem `for`, tedy odpovídá hodnotě 1.

Přestože se tedy u obou částí gramatiky ukázázalo, že jejich odstranění může vést k nejasnostem, faktem zůstává, že u valné většiny kódu nejsou nutné. Nakonec tedy překladač oboje podporuje, nicméně s varováním a jde na ně nahlížet jako na formu zotavení z chyb. Podmínky jsou řešeny výjimkou, kdy pokud začínají `(` nebo `!(` jsou zpracovány konvenčně, v opačném případě s varováním jako obecný výraz.

Kapitola 6

Implementace

Pro implementaci překladače byl zvolen jazyk C, což má kromě rychlosti i další praktický důvod – pro překlad překladače není nutný vyšší jazyk a překladač C je pro praktické použití C+ stejně nutností. Vzhledem k tomu, že překladač využívá jen standardní knihovny C, je efektivně C+ použitelné na stejné množině platforem jako C.

Program má základní rozhraní, jako parametry očekává jména zdrojových souborů, v cyklu pro každý inicializuje datové struktury, provede syntaktickou a sémantickou analýzu spojenou s překladem rozšiřujících aspektů C+ do čistého C a pokud nenastala závažná chyba, provede výpis výsledného kódu na standardní výstup.

Vzhledem k tomu, že při programování v C je často mnoho chyb spojených s dynamickou alokací paměti, bylo klíčové navržení optimálních datových struktur. Identifikátory, literály, datové typy, funkce a výrazy jsou ukazateli do příslušných tabulek, které tvoří jediné perzistentní dynamické datové struktury.

Tabulky mají formu jednosměrných seznamů polí. Jednotlivé položky se zabírají postupně, v případě zaplnění se alokuje pole nové. Předčasné uvolnění nepoužívaných položek není nutné, pro obsah s omezenou platností se použijí lokální tabulky.

Tabulka identifikátorů obsahuje pouze dynamicky alokované C řetězce bez sémantiky, každý identifikátor je však unikátní – porovnávání idenfitikátorů v programu pak má konstantní složitost. Tabulka literálů obsahuje textové řetězce i číselné literály, oboje uložené také jako dynamicky alokované C řetězce, neboť v rámci překladu není nutné nijak vyhodnocovat jejich hodnotu, je však nutné znát jejich datový typ, který je v jednotlivých záznamech uložen. Tabulky datových typů obsahují typové základy[1] se seznamem kvalifikovaných základů. Kvalifikované základy pak obsahují seznam konkrétních typů, lišících se případným seznamem ukazatelů, polí či funkcí. První položky seznamů jsou umístěny lokálně. Tabulka funkcí obsahuje jednotlivé funkční základy s unikátním jménem a návratovým typem, které obsahují seznam konkrétních přetížení. Každé přetížení jestě může obsahovat seznam subvariant. Tabulka výrazů obsahuje jednotlivé výrazové bloky, vysvětlené dále.

Výrazy jsou uloženy jako binární strom výrazových bloků, kde kazdý blok obsahuje typ operátoru, výsledný datový typ a jeden nebo dva ukazatele, ukazující podle kontextu na podblok, identifikátor, literál, datový typ nebo funkci – např. výraz $a = 3 + 5$ je složen pouze z jednoho bloku přiřazení a jednoho bloku sčítání.

Kód je složen z hierarchického obousměrného seznamu kódových bloků odpovídající typům příkazů a deklarací. Všechny typy bloků sdílejí základní ukazatele a rozšiřují základ

[1] pro typy interní, vlastní, výčtové i kompozitní

o vlastní položky, např. blok `if` obsahuje výraz v podmínce a ukazatele na podseznamy v if a else větvi. Lokální tabulky typů a funkcí jsou uloženy v kódovém bloku typu podblok.

Toto rozložení je velmi efektivní pro vysokoúrovňové transformace kódu, samotný překlad do čistého C je triviální implementace postupů z předchozích kapitol. Implementačně složitá je naopak syntaktická a sémantická analýza.

Lexikální analyzátor pracuje na principu konečného automatu. Čtení ze vstupu je striktně lineární, největší nutný náhled dopředu pro rozeznání tokenu je 2 znaky[2], které jsou uloženy ve čtecí struktuře. U literálů je odvozen datový typ, u identifikátorů jsou rozeznány klíčová slova pomocí několikaúrovňové konstrukce `switch`.

Syntaktická analýza používá rekurzivní sestup odpovídající gramatickým pravidlům. Mezi funkcemi je předáván kontext ve struktuře analyzátoru, který kromě čtecí struktury obsahuje seznam vytvářených kódových bloků a informace o odsazení pro odsazování bílými znaky. Dynamicky alokovaná data (jako seznam argumentů funkce) jsou buď instantně přidána do tabulek, nebo dealokována ještě na stejné úrovni. Duplicity v gramatice (jako ty vyniklé problémy s návěstím zmíněným v 4.6) byly vyřešeny náhledem o jeden token dopředu na sporných místech. Nad lexikálním analyzátorem je mezivrstva, která u tokenu nového řádku počítá odsazazení dalšího řádku a v případě potřeby generuje token konce bílými znaky odsazeného bloku.

Sémantické kontroly jsou prováděny podle potřeby – na místech, kde jsou přijímány identifikátory je nalezen jejich lokální význam. Po syntaktické analýze výrazu jsou rekurzivně odvozeny datové typy podvýrazů, je vyřešeno i přetížení. Ze subvariant je nejprve vybrána ta vracející datový typ, při druhém průchodu jsou u volání, kde není výsledek využit, použity void subvarianty. Výpis výsledného kódu je pak jen prostým průchodem kódovými bloky s rekurzivním výpisem obsahu.

[2] pro rozeznání … od .

Kapitola 7

Závěr

Byla proveden návrh rozšíření jazyka C, vycházející z analýzy výhod řady vyšších programovacích jazyků, s kanonickým překladem rozšiřujiécích vlastností do C. Pro tento nově vytvořený jazyk byla sestrojena formální gramatika třídy LL(1) i s LL tabulkou. Na závěr byl úspěšně implementován překladač daného jazyka do čistého C.

Nový jazyk nabízí prostředky k psaní kratšího a přehlednějšího kódu, oproti jiným srovatelným populárním jazykům ale zásadně nekompromituje výkonnost ani neodstiňuje uživatele od vnitřní implementace, čímž nabízí zajímavou alternativu. Relativní vzácnost podobných řešení otevírá pole pro další perspektivní vývoj.

Přestože byl cíl splněn, existuje řada neřešených aspektů, které dávají podněty k budoucí práci.

Nebyl řešen preprocessing jazyka C, zejména tedy rozgenerování maker, kondicionální překlad a direktiva `include`. Mnoho překladačů C umožňuje provést samostatný preprocessing, konceptuálně tak toto není překážkou, problémem je, že při extérním řešení `include` nelze poznat, jaké funkce pochází z čistého C – u všech je provedeno zakódování pro přetížení, což znemožnuje linkování s C moduly, primárně knihovnami.

Dále je tu několik neimplementovaných částí jazyka. Kromě některých sémantických kontrol jde např. o takzvaný „starý styl“ definic funkcí s externími deklaracemi typů. Mimo to, že je redundantní a i v samotném standardu C označen jako zastaralý, také díky němu vede chybějící středník za deklarací funkce k nelokální chybě a znemožňuje tím volitelnost stedníku na daném místě. Dále jde o podporu digrafů a trigrafů, podpora vlastností z novějšího standardu C11, případně snad i některých GNU C rozšíření.

Kromě práce na překladači lze v budoucnu také realizovat další rozšíření jazyka C+, která doplňují a rozvíjejí ty zde implementované. Následuje stručný výběr z již konceptuálně vyřešených částí:

- *obsluha vyjímek*
  Řeší problém ošetřování chyb v přetížených operátorech a přináší strukturovaný způsob ošetřování chyb. Funkce, která si nechá ošetřovat vyjímečný stav, je považována za další subvariantu a má skrytý parametr booleovský parametr signalizující chybu.

- *konstruktory a destruktory*
  Vhodné zejména pro držení zdrojů jako je dynamická paměť či soubory, tedy pro filosofii RAII (Resource Aquisition is Initialization). Minimalistická implementace přidávající volání určených metod na místě deklarace, konci bloku a výskoku z bloku.

Řešena chyba v návrhu gramatiky C++, kde je deklarace funkce někdy nerozeznatelná od deklarace proměnné s konstruktorem, jako např. `List f(String ());`

- *funkční aliasy*
  Funkce definované jako jíné funkce, vhodné např. pro použití funkce jako jedno přetížení jiné funkce nebo jako metodu typu. Implementace definováním makra.

- *(L-hodnotové) reference*
  Vhodné pro vytvoření funkcí vracející L-hodnotu. Na rozdíl od C++, kde jsou definovány sémanticky a implementace je na překladači, jsou v C+ přesně definovány jako implicitně dereferencované ukazatele, jde je tedy kombinovat s čistým C.

- *cyklus foreach*
  Iterativní cyklus `for`. Přidán podobné jako v C++11, byť se syntaxí bližší C#. Převeden na normální `for`, vyžaduje po použitém datovém typu existenci vhodných metod.

Literatura

[1] The linux-kernel mailing list FAQ. [online], Poslední modifikace: 17. října 2010 [vid. 18. dubna 2015] Dostupné z: www.tux.org/lkml/#s15.

[2] Python 3 language reference. [online], Poslední modifikace: 5. května 2015 [vid. 8. května 2015] Dostupné z: docs.python.org/3/reference/index.html.

[3] Přechod GCC na rekurzivní sestup. [online], Poslední modifikace: 1. října 2008 [vid. 9. května 2015] Dostupné z: www.agner.org/optimize/calling_conventions.pdf.

[4] Standard jazyka C99 – ISO International Standard ISO/IEC 9899:TC3:2011. [online], [vid. 21. listopadu 2014] Dostupné z: www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[5] Aho, A.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools.* Boston: Pearson Education, druhé vydání, 2007, iSBN 0-321-48681-1.

[6] Cooper, K.; Torczon, L.: *Engineering a compiler.* London: Morgan Kaufmann, 2004, iSBN 80-01-01428-2.

[7] Dijkstra, E.: Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, ročník 11, č. 3, Březen 1968: s. 147–148, ISSN 0001-0782.

[8] Fog, A.: Calling conventions for different C++ compilers and operating systems. [online], Poslední modifikace: 7. srpna 2014 [vid. 9. května 2015] Dostupné z: www.agner.org/optimize/calling_conventions.pdf.

[9] Holub, A.: *Compiler design in C.* London: Prentice Hall, 1990, iSBN 0-13-155045-4.

[10] Meduna, A.: *Automata and Languages: Theory and Applications.* London: Springer Verlag, 2005, iSBN 1-85233-074-0.

[11] Meduna, A.: *Elements of Compiler Design.* New York: Taylor & Francis, 2008, iSBN 1-4200-6323-5.

[12] Rychta, K.; Sochor, J.: *Softwarové inženýrství I.* Praha: Vydavatelství ČVUT, 1996, iSBN 80-01-01428-2.

[13] Sebesta, R. W.: *Concepts of programming languages.* London: Addison-Wesley, 10 vydání, 2012, iSBN 0-13-139531-9.

[14] Stroustrup, B.: *Standard C++.* Chichester: J. Wiley, 2003, iSBN 0-470-84674-7.

[15] Wirth, N.: *Compiler construction*. Harlow: Addison-Wesley, 1996, iSBN 0-201-40353-6.

Dodatek A

Obsah CD

Přiložený disk obsahuje zdrojové kódy překladače v adresáři `src`. Překlad lze provést pomocí přiloženého souboru Makefile, případně prostým přeložením a slinkováním všech modulů s příponou c za použití standardu C99.

Ve složce `bin` je překladač ve formátu exe. Použití je jednoduché, argumenty programu jsou chápány jako jména souborů, výstup a chybový výstup je standardní. Žádné přepínače nejsou v současnosti podporovány.

Dále je tu text práce v pdf i ve zdrojové formě v adresářích `doc` a `docsrc`.

Ve složce `grammar` se nachází LL gramatika a LL tabulka jazyka. Mimo ně je zde přiložen program v Pythonu, kterým byla tabulka vytvořena.

Konečně, ve složce `examples` se nachází přeložitelné příklady v jazyce C+, ilustrující různé vlastnosti jazyka. Příklady simplefilter a parser jsou výsledkem preprocesoru C spuštěným na kompletně přeložitelných C modulech.