Sem vložte zadanie Vašej práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Teoretické Informatiky

Bakalárska práca

Přední část překladače GCC pro vnitřní formu GCC

Martin Senko

Vedúci práce: Ing. Jan Trávníček

11. februára 2015

Poďakovanie

Touto cestou by som sa chcel poďakovať Ing. Janovi Trávníčkovi za jeho čas, ochotu a vynaložené úsilie pri našej spolupráci.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 11. februára 2015 .....

Odkaz na túto prácu

Senko, Martin. *Přední část překladače GCC pro vnitřní formu GCC.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Táto práca analyzuje jeden zo vstavaných výstupných formátov vnútornej reprezentácie programu v GCC a informačnú hodnotu, ktorú obsahuje. Práca ďalej obsahuje návrhy na jeho rozšírenie a popisuje implementáciu prednej časti prekladača, ktorá príjma daný výstupný formát a realizuje jeho preklad do vnútornej reprezentácie GENERIC.

Klíčová slova predná časť prekladača, vnútorná reprezentácia, prekladač GCC

Abstract

This thesis describes and analyzes built-in intermediate representation dump format of GCC compiler and the informational value contained. It suggests extensions to the dump format, making it suitable for further front-end processing. This thesis also describes the front-end which was developed for the dump format processing and translating into GENERIC intermediate representation.

Keywords compiler front-end, intermediate representation, GCC compiler

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

GCC je obľúbeným prekladačom jazyka C/C++. Od čias, kedy šlo len o prekladač jazyka C sa toho veľa zmenilo a GCC dnes podporuje ďalšie jazyky ako Ada, Fortran, Go a iné.

GCC bol od začiatku navrhovaný ako multijazyčný a multiplatformný prekladač. Aj vďaka tomu je dnes možné prekladač rozšíriť o novú prednú časť, ktorá sa postará o preklad vstupného kódu do vnútornej reprezentácie GCC, bez nutnosti budovať kompletnú strednú a koncovú časť prekladača. Keď predná časť odovzdá program reprezentovaný vo vnútornej forme GIMPLE, stredná a zadná časť sa postará o jeho preklad do objektového kódu pre vybranú cieľovú architektúru, bez ďalších potrebných zásahov do týchto častí.

Plne funkčná predná časť, ktorá by spracovávala výstup niektorej z vnútorných reprezentácií prekladača, mi zatiaľ nie je známa[1]. Takáto predná časť by prispela k porozumeniu interných procesov v prekladači a umožnila by rýchlejšie hľadanie a opravu chýb, predovšetkým pri návrhu nových predných častí. Mohla by tiež slúžiť na vzdelávacie účely.

[1] Za zmienku určite stojí GIMPLE FrontEnd. Ide o koncept prednej časti, ktorá by spracovávala n-tice vnútornej reprezentácie GIMPLE. Podľa dostupných informácii však projekt nedospel do štádia, kedy by bola predná časť odladená a plne funkčná. Aj napriek tomu však tento projekt svedčí o záujme o podobný nástroj.

Kapitola 1

Cieľ práce

V tejto práci sa budem zaoberať predovšetkým analýzou výstupného formátu vnútornej reprezentácie GENERIC a tvorbou prednej časti prekladača, ktorá bude spracovávať jej textovú formu a následne prevádzať preklad do vnútornej reprezentácie. Po rozšírení by mal byť prekladač, za pomoci implementovanej prednej časti, schopný preložiť výstup vnútornej reprezentácie do cieľového kódu. Výsledkom prekladu bude spustiteľný súbor s funkcionalitou danou pôvodným vstupným kódom.

Predná časť by malá byť schopná preložiť základnú množinu uzlov reprezentácie GENERIC tak, aby bolo možné overiť funkčnosť navrhovaného konceptu[2]. Pri vytváraní prednej časti preto budem uvažovať nasledovné:

- deklarácie funkcií
- konštanty
- skalárne premenné
- polia
- základné aritmetické a logické výrazy
- volanie funkcií (aj externé)
- cykly *for* a *while*
- podmienky

Predná časť prekladača bude pri spracovaní vstupu kontrolovať jeho syntaktickú validitu, a upozorňovať na prípadné chyby. Predná časť bude prevádzať nasledujúce kontroly:

[2]Na testovanie bude použitý jazyk C. Podporované uzly preto zodpovedajú základným konštrukciám v jazyku C. Iné jazyky môžu využívať na reprezentáciu základných jazykových konštrukcií uzly, ktoré implementovaná predná časť nemusí podporovať.

- kontrola odkazu (odkazovať je možné len na uzol, ktorý je v zdrojovom súbore deklarovaný)
- kontrola poradia (uvažujeme striktný formát zdrojových súborov, kde sú jednotlivé uzly označované vo vzostupnom poradí, s krokom 1)
- kontrola typu uzla (uzol musí byť z množiny podporovaných uzlov)
- kontrola parametrov uzla (parameter musí byť z množiny prípustných parametrov uzla)
- kontrola typu hodnoty parametra (parameter má striktne definovaný typ hodnoty, ktorú môže nadobúdať)

Kapitola 2

Analýza a návrh riešenia

2.1 História GCC

GCC napísal Richard Stallman ako prekladač jazyka C (GNU C Compiler). Prvýkrát bol vydaný dňa 22. marca 1987.

Napriek tomu, že šlo o prekladač jazyka C, Stallman od začiatku pripravoval GCC pre podporu viacerých jazykov a cieľových platforiem.

Pôvodne mal prekladač GCC vychádzať z prekladača Pastel, no po tom čo si Stallman uvedomil, že Pastel v rámci syntaktickej analýzy ukladá celý vstupný súbor do AST, ktorý následne konvertuje do reťazca inštrukcií, z ktorých nakoniec generuje výstupný súbor bez akéhokoľvek uvoľnovania prostriedkov, rozhodol sa začať od začiatku.

Vývoj prekladača (dovtedy Cygnus a FSF) sa zdal byť mnohým zúčastneným príliš konzervatívny, a tak niektorí z nich začali s vývojom vlastných vetiev. Niekoľko takýchto ľudí sa v roku 1977 spojilo, aby vytvorili projekt EGCS.

V apríli 1999 sa po dlhom vyjednávaní stal projekt EGCS oficiálnou verziou GCC. V tomto čase sa zmenil aj význam skratky GCC na GNU C Compiler. Prvý prekladač, vydaný po tejto významnej zmene, bol GCC verzie 2.95.

Dnes sa vývoj GCC riadi presne definovaným plánom vývoja [1] a podlieha vedeniu riadiaceho výboru.

GCC obsahuje jazyky C, C++, Objective-C, Fortran, Java, Ada a Go. Súčasťou sú aj knižnice pre tieto jazyky. Vyššie uvedené informácie sú prevzaté z oficiálnych zdrojov [2].

2.2 Zostavenie prekladača

Zdrojové súbory potrebné k zostaveniu prekladača sú voľne dostupné z rôznych lokalít, ktorých zoznam sa nachádza na stránkach projektu GCC [3], alebo priamo z projektovej SVN.

Po rozbalení archívu je potrebné vytvoriť nový adresár, kam sa bude GCC zostavovať. Zostavenie do zdrojového adresára nie je podporované. Tento scenár nie je v rámci vývoja testovaný, preto môže dôjsť k chybám. Výhodou prekladu do samostatného adresára je predovšetkým to, že sa tým nezasahuje do zdrojových súborov prekladača. Vďaka tomu je v prípade potreby kedykoľvek možné celý prekladač zostaviť znova, bez nutnosti sťahovať zdrojové súbory.

```
cd gcc-src
./contrib/download_prerequisites
cd ..
mkdir gcc-obj
cd gcc-obj
../gcc-src/configure --disable-multilib --enable-checking
--enable-languages=rawtree
make
```

2.2.1 Konfigurácia a zostavenie

Konfiguračný skript je potrebné spustiť z vopred vytvoreného adresára (v tomto prípade `gcc-obj`). V tom istom adresári následne zostavíme prekladač príkazom `make`.

GNU Make od verzie 3.80 podporuje paralelné zostavovanie. Pre využitie tejto možnosti je potrebné pridať k príkazu `make` prepínač `-j n`, kde n špecifikuje počet *receptov* zostavovaných súčasne.

V práci budem uvažovať vyššie uvedenú adresárovú štruktúru. Pokiaľ nebude uvedené inak, všetky cesty k zdrojovým súborom budú uvádzane z adresára `gcc-src`.

2.2.1.1 Kontrola konzistencie

Prepínač `--enable-checking` povoľuje kontrolu konzistencie v rámci štruktúr GCC. Tieto kontroly výrazne uľahčujú prácu s vnútornými reprezentáciami GCC, pretože prekladač prevádza typovú kontrolu nad rámec základných typov a pri chybe je z chybového výstupu prekladača možné získať pomerne detailné informácie o typovej nekonzistencii. Ak sú pri zostavení prekladača použité zdrojové súbory z oficiálnej SVN projektu, kontrola konzistencie je automaticky povolená.

Kontrola konzistencie je prevádzaná v dobe behu prekladača a jej réžia je značná. Pre bežné zostavenie prekladača sa preto odporúča túto možnosť vypnúť buď úplne, alebo je možné vypnúť len niektoré jej kategórie [4].

2.2.1.2 Znovuzostavenie prekladača

Každé ďalšie zostavenie GCC pri budovaní prednej časti prevedieme príkazom `make`, z adresára `gcc-obj`. Kvôli prípadným chybám a upozorneniam, je vhodné presmerovať chybový výstup programu `make` do súboru, a neskôr z neho prípadne filtrovať požadované informácie (napríklad len chyby).

2.3 Časti prekladača

2.3.1 Predná časť

Úlohou prednej časti prekladača je previesť vstup na vnútornú reprezentáciu GCC. Práca prednej časti prekladača býva často rozdelená do funkčných celkov. Najprv je zvyčajne prevádzaná lexikálna analýza vstupu (spracovanie vstupu a jeho prevod na postupnosť tokenov). Nad postupnosťou tokenov sa potom prevádza syntaktická analýza. Preklad potom väčšinou nie je prevádzaný priamo do vnútornej reprezentácie, ale do vlastného AST, ktorý tvorí akýsi medzičlánok medzi vstupným programom a vnútornou reprezentáciou GCC, čo zjednodušuje ďalší preklad. Z neho sa potom prevádza preklad do samotnej vnútornej reprezentácie GCC. V rámci prekladu sa predné časti tiež starajú o kontrolu sémantiky. Po preklade do vnútornej reprezentácie GENERIC sa ešte v rámci prednej časti prevádza zjednodušenie na reprezentáciu GIMPLE. Program v tejto forme je potom odovzdaný na ďalšie spracovanie strednej časti.

2.3.2 Stredná časť

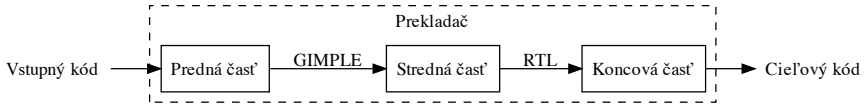
Stredná časť prijíma program reprezentovaný ako zjednodušené stromy vnútornej formy GIMPLE a prevádza nad nimi optimalizácie na úrovni stromov. Výstupom strednej časti prekladača je program reprezentovaný v jazyku RTL.

2.3.3 Koncová časť

Koncová časť prijíma program reprezentovaný v jazyku RTL a prevádza nad ním nízkoúrovňové optimalizácie. Po prevedení optimalizácií sa generátor kódu stará o generovanie objektového kódu.

2.4 Vnútorná reprezentácia programu v GCC

Zjednodušene povedané, GCC pracuje počas prekladu s dvomi vnútornými reprezentáciami. Prvou sú stromové štruktúry a druhou je jazyk RTL. Všetky predné časti GCC prekladajú vstupný kód do jazykovo nezávislej vnútornej reprezentácie v GCC. Kým v minulosti predstavoval túto nezávislú reprezentáciu programu jazyk RTL, dnes je k dispozícii reprezentácia vyššej úrovne – GENERIC.



Obr. 2.1: Základné časti prekladača GCC

2.4.1 GENERIC

Vnútorná reprezentácia GENERIC [5, Kapitola 10] slúži ako spoločný, jazykovo nezávislý vstupný bod pre predné časti prekladača. Vznikla ako nadmnožina všetkých existujúcich reprezentácii stromov v GCC. Predné časti teraz prevádzajú svoj kód do reprezentácie GENERIC.

2.4.1.1 Rozhranie

Na preklad a manipuláciu s reprezentáciou GENERIC poskytuje GCC rozsiahle rozhranie, definované v súbore `gcc/tree.h`. Vnútorná štruktúra jednotlivých typov stromov je následne definovaná v súbore `gcc/tree-core.h`. V tomto súbore nájdeme aj definíciu samotného typu `tree`, ktorý je definovaný ako *union* všetkých prípustných štruktúr stromu. Typy uzlov, ich kódy (`tree_code`) a počet operandov je možné nájsť v súbore `gcc/tree.def`.

2.4.1.2 Štruktúra

Základná dátová štruktúra vnútornej reprezentácie GENERIC je `tree`. Zatiaľ čo všetky uzly sú typu `tree`, v skutočnosti môžu predstavovať rôzne varianty tohto typu. Každý `tree` je ukazovateľ, no objekt, na ktorý ukazuje, môže byť rôzneho typu. Napriek tomu, že väčšina funkcií a makier pracuje s obecným typom `tree`, vo vnútri GCC existuje typový systém pre stromy, ktorý nie je súčasťou typového systému C a ten umožňuje kontrolovať typ stromu. Vďaka tomu sa dá množstvo užitočných funkcií a makier použiť len na vybrané typy stromov.

Špeciálnym stromom je `error_mark_node`, ktorého kód je `ERROR_MARK`. Takýto strom však môže existovať len jeden, preto sa často v porovnaniach vyskytuje priamo `error_mark_node` (porovnanie na rovnosť ukazovateľov). Ak počas spracovávania vstupu prednou časťou nastala chyba, nastaví sa flag `errorcount`. Pokiaľ predná časť narazí na kód, ktorý nevie spracovať, nastaví flag `sorrycount`. Po nastavení týchto flagov by mali všetky makrá a funkcie, ktoré za normálnych okolností vracajú strom príslušného typu, vracať `error_mark_node`.

2.4.1.3 Makrá a funkcie

V rámci konvencie sú názvy makier definované veľkými písmenami, zatiaľ čo názvy funkcie pozostávajú výhradne z malých písmen. Je však možné naraziť aj na výnimky. Dôležité však je, že každé makro alebo funkcia, ktorej názov pozostáva výhradne z veľkých písmen, môže vyhodnocovať svoje argumenty viac ako raz. Makrá a funkcie s názvami tvorenými malými písmenami vyhodnocujú svoje argumenty len raz.

Na zistenie typu uzla slúži makro `TREE_CODE`, vďaka ktorému z uzla získame jeho `tree_code`, tak ako je uvedený v `gcc/tree.def`. Túto informáciu obsahujú všetky uzly. Funkcia `tree_size` vracia veľkosť stromu v bytoch.

`build0 - build6` sú univerzálne funkcie, pomocou ktorých je možné vytvárať stromy. Na vytvorenie stromu zvoleného typu je potrebné použiť správnu funkciu `buildn`, kde n je počet operandov, ktoré daný typ stromu potrebuje poznať (okrem `TREE_CODE` a `TREE_TYPE`). Tento údaj je uložený ako posledná (4.) položka v definícii jednotlivých uzlov, v súbore `gcc/tree.def`. Parametre pri volaní funkcie `buildn` budú odovzdané nasledovne:

```
buildn (code, type, [operands]')
```

Pre funkciu `build0` to teda bude len kód stromu a typ.

2.4.2 GIMPLE

Pri preklade programu z jazykovo závislých stromových reprezentácií do vnútornej reprezentácie GENERIC sa odstraňujú jazykové závislosti, no štruktúra stromov, ktoré reprezentujú program, je stále príliš zložitá z hľadiska optimalizácií. Na jej zjednodušenie sa používa reprezentácia GIMPLE [5, Kapitola 11].

Reprezentácia GIMPLE je odvodená od GENERIC a zjednodušenie spočíva predovšetkým v rozdelení výrazov reprezentácie GENERIC do n-tíc, ktoré majú najviac 3 operandy (existujú výnimky). GIMPLE navyše obsahuje dočasné premenné, ktoré sú potrebné pre výpočet zložitejších výrazov, ako v nasledujúcom príklade.

GENERIC	GIMPLE

`a = b++ * c`	`tmp = b`
	`b = b + 1`
	`a = tmp * c`

GCC má v sebe vstavaný priechod, nazvaný `gimplifier`, ktorý zabezpečuje konverziu z reprezentácie GENERIC do GIMPLE. Gimplifier pracuje rekurzívne a n-tice reprezentácie GIMPLE generuje z pôvodných GENERIC výrazov. Za ťažisko práce preto budem považovať preklad do vnútornej reprezentácie GENERIC. Následná transformácia do GIMPLE je totiž prevedená volaním vstavanej funkcie na koreňové uzly stromov.

2.5 Vstavaný výstup vnútornej reprezentácie

Z prekladača GCC je možné nechať počas prekladu vygenerovať výstup, ktorý je textovou reprezentáciou danej vnútornej formy. K dispozícií je viacero možností. Prepínač pre aktiváciu generovania výstupu má jeden z nasledujúcich tvarov:
`-fdump-tree-`*switch*
`-fdump-tree-`*switch*`-`*options*
`-fdump-tree-`*switch*`-`*options*`=`*filename*
kde *switch* predstavuje zvolenú vnútornú reprezentáciu a *options* sú možnosti, ktoré je možné pri generovaní výstupu uplatniť. Ich kompletný zoznam a popis sa nachádza v dokumentácii, na stránkach projektu [6]. Nakoniec je možné explicitne definovať meno súboru, do ktorého sa textová reprezentácia vygeneruje.

Pre účely tejto práce bol použitý formát `original-raw`, pretože poskytuje komplexné informácie o jednotlivých uzloch, a je tak možné prekladať ho do vnútornej reprezentácie *postorder* priechodom, vygenerovaným stromom. Tento výstup je generovaný v nasledujúcom tvare:

```
@n  tree_code  par0: @x  par1: y  par2: string
```

Úvodné `@`n predstavuje celočíselné označenie uzla v rámci konkrétnej deklarácie funkcie. Týmto číslom sa naň odkazujú iné uzly. Nasledujúci reťazec reprezentuje kód uzlu (`tree_code`). Pod týmto označením ho môžeme nájsť v súbore `tree.def`.

Nasledujú `par0` - `par`n, ktoré predstavujú jednotlivé parametre daného uzla. Ich názvy väčšinou pozostávajú zo 4 písmen a sú odvodené od významu daného parametra (napr. `type` pre typ, `prec` pre presnosť, alebo `algn` pre zarovnanie). V niektorých prípadoch sú použité len 3 (`int`, `min`, ...), prípadne 2 (`fn`) písmená. Uzly, ktoré obsahujú operandy, sú generované ako parametre s uniformným názvom `op 0` - `op` n. Špeciálnym prípadom je uzol `statement_list`, ktorého položky sú generované ako číslovaný zoznam od 0 po n. Hodnoty parametrov môžu byť trojakého typu. Prvým typom je odkaz na iný uzol (`@`n), druhým je celé (prípadne desatinné) číslo, a posledným reťazec.

Na základe vyššie uvedených poznatkov môžeme formalizovať gramatiku pre jazyk, daný výstupným formátom `original-raw`. Formálne zapísaná gramatika má nasledovný tvar:
$G = (N, T, P, S)$
$N = \{Nodes,\ Nodes',\ Node,\ NodeArgs,\ NodeArg,\ Arg,\ Val,\ LineNo\}$
$T = \{\text{RPTR, STR, STR\_RAW, NUMB, :, op}\}$
$S = Nodes$

Množina P obsahuje nasledujúce pravidlá:

1. $Nodes \rightarrow Node\ Nodes'$
2. $Nodes' \rightarrow Node\ Nodes'$
3. $Nodes' \rightarrow \varepsilon$
4. $Node \rightarrow$ RPTR STR $NodeArgs$
5. $NodeArgs \rightarrow NodeArg\ NodeArgs$
6. $NodeArgs \rightarrow \varepsilon$
7. $NodeArg \rightarrow Arg : Val$
8. $Arg \rightarrow$ STR
9. $Arg \rightarrow$ op NUMB
10. $Arg \rightarrow$ NUMB
11. $Val \rightarrow$ RPTR
12. $Val \rightarrow$ STR $LineNo$
13. $Val \rightarrow$ STR_RAW
14. $LineNo \rightarrow$: NUMB
15. $LineNo \rightarrow \varepsilon$

Na základe uvedenej gramatiky môžeme vypočítať hodnoty funkcií *FIRST* a *FOLLOW* pre pravé strany pravidiel gramatiky 2.1.
Gramatika je $LL(1)$, takže môžeme zostaviť rozkladovú tabuľku 2.2, ktorá je deterministická.

2.5.1 Analýza informácií obsiahnutých vo výstupe

Vstavaný výstup `original-raw` nie je vhodný na priamu rekonštrukciu programu, pretože neobsahuje kompletné informácie o reprezentovanom programe. Poskytuje však dobrý základ, a po prevedení malých úprav je z neho možné program rekonštruovať.

Jedným z problémov výstupu je, že funkcia generujúca výstup je volaná rekurzívne nad uzlom, ku ktorému pristupuje makrom `DECL_SAVED_TREE` v deklarácii funkcie. Počiatočným uzlom je preto vždy `STATEMENT_LIST`, ktorý obsahuje dva uzly. Prvý, `BIND_EXPR`, spája blok funkcie s jeho lokálnymi premennými a druhý, `RETURN_EXPR`, predstavuje návratovú hodnotu. Z pohľadu hierarchie v AST, `STATEMENT_LIST` nie je koreňovým uzlom funkcie (obrázok 2.2). Výhodnejšie bude začať priamo uzlom `FUNCTION_DECL`, ktorý reprezentuje deklaráciu funkcie.

Tabuľka 2.1: Množiny FIRST a FOLLOW pre pravé strany pravidiel

#	Pravidlo	FIRST	FOLLOW
1	$Nodes \rightarrow Node\ Nodes'$	RPTR	ε
2	$Nodes' \rightarrow Node\ Nodes'$	RPTR	ε
3	$Nodes' \rightarrow \varepsilon$	ε	
4	$Node \rightarrow$ RPTR STR $NodeArgs$	RPTR	ε, RPTR
5	$NodeArgs \rightarrow NodeArg\ NodeArgs$	STR, op, NUMB	ε, RPTR
6	$NodeArgs \rightarrow \varepsilon$	ε	
7	$NodeArg \rightarrow Arg : Val$	STR, op, NUMB	ε, RPTR, STR, op, NUMB
8	$Arg \rightarrow$ STR	STR	:
9	$Arg \rightarrow$ op NUMB	op	
10	$Arg \rightarrow$ NUMB	NUMB	
11	$Val \rightarrow$ RPTR	RPTR	ε, RPTR, STR, op, NUMB
12	$Val \rightarrow$ STR $LineNo$	STR	
13	$Val \rightarrow$ STR_RAW	STR_RAW	
14	$LineNo \rightarrow$: NUMB	:	ε, RPTR, STR, op, NUMB
15	$LineNo \rightarrow \varepsilon$	ε	

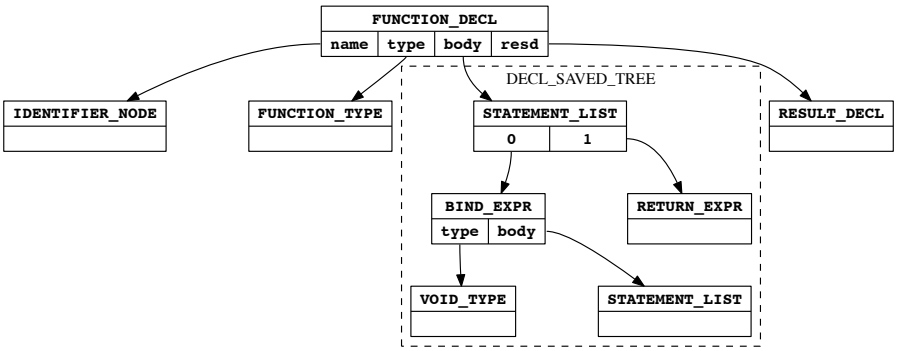
Tabuľka 2.2: Rozkladová tabuľka gramatiky

	RPTR	STR	STR_RAW	NUMB	op	:	ε
Nodes	1						
Nodes'	2						3
Node	4						
NodeArgs	6	5		5	5		6
NodeArg		7		7	7		
Arg		8		10	9		
Val	11	12	13				
LineNo	15	15		15	15	14	15

Ďalší problém pri spracovaní deklarácie funkcie je absencia uzla `RESULT_DECL` vo výstupe. Prístup k tomuto uzlu v rámci deklarácie funkcie zabezpečuje makro `DECL_RESULT`. V tomto prípade postačí rozšíriť výstup o tento uzol.

Z parametrov uzla `DECL_EXPR` nie je možné určiť, ku ktorej deklarácii premennej daný deklaračný výraz patrí. Prístup k príslušnej deklarácii premennej zabezpečuje makro `DECL_EXPR_DECL`, takže tiež postačí rozšíriť výstup o daný uzol.

Deklarácie premenných v rámci bloku sú v GCC principiálne reprezentované ako jednosmerný spojový zoznam. Vo výstupe však chýba nasledujúca deklarácia premennej v prípade, že ich je v bloku deklarovaných viac. Do výstupu bude potrebné pridať odkaz na nasledujúcu deklaráciu premennej.



Obr. 2.2: `DECL_SAVED_TREE` v kontexte deklarácie funkcie

Výstup textových reťazcov sa do súboru generuje bez špeciálneho znaku na začiatku a na konci reťazca. Kvôli bielym znakom preto $LL(1)$ syntaktickou analýzou nevieme rozhodnúť, či má byť reťazec považovaný za hodnotu alebo parameter. Aby sme mohli využiť $LL(1)$ syntaktickú analýzu, bude potrebné reťazec obaliť do špeciálnych znakov.

Špeciálne znaky boli pridané aj ku reťazcom, ktoré reprezentujú názvy zdrojových súborov (na výstupe parameter `srcp`). Vďaka tomu sa zjednodušil lexikálny analyzátor a zároveň zväčšila množina prípustných znakov v názve súboru.

2.5.1.1 Globálne premenné

Informácie o globálnych premenných nie sú vo výstupe obsiahnuté vôbec. Pri použití globálnej premennej v tele funkcie sa však v strome danej funkcie objaví uzol `VAR_DECL`, ktorý neobsahuje parameter `scpe`. Inak povedané, takýto uzol nemá definovaný rozsah platnosti, takže ho možno považovať za globálnu premennú. V prípade potreby sémantickej analýzy by však nebolo možné overiť, či bola požadovaná premenná deklarovaná, pretože najvyššia úroveň rozsahu platnosti, s ktorou v rámci vnútornej reprezentácie GCC pracujeme, je funkcia.

Kapitola 3

Realizácia

Všetky úpravy ako aj samotná implementácia prednej časti boli prevedené na prekladači GCC verzie 5.0.0 20141025 (experimental). Pre vytvorenie rovnakých podmienok je možné stiahnuť zdrojové súbory prekladača v rovnakej revízii (rev. 216689), z repozitára na `svn://gcc.gnu.org/svn/gcc`.

3.1 Úprava výstupu

Generovanie výstupu začína volaním funkcie `c_genericize` v súbore `gcc/c-family/c-gimplify.c`, odkiaľ pochádza hlavička každej funkcie. Logika generovania informácii o jednotlivých uzloch obsiahnutých vo výstupe `-original-raw` je potom implementovaná v súbore `gcc/tree-dump.c`.

Vo funkcii `c_genericize` som upravil parameter počiatočného volania funkcie `dump_node`. Táto funkcia pôvodne začínala priechod v uzle, dostupnom z koreňového uzla `FUNCTION_DECL` makrom `DECL_SAVED_TREE`. Po úprave priechod a výpis začína priamo uzlom `FUNCTION_DECL` príslušnej spracovávanej funkcie. Uzly sú vypisované na výstup v poradí *preorder*, takže sa pri implementácii prednej časti budem opierať o fakt, že prvým uzlom v rámci funkcie je vždy jej deklarácia.

Do výstupu deklarácie funkcie som tiež pridal uzol `RESULT_DECL`, ktorý predstavuje výsledok (návratovú hodnotu) funkcie. Rekurzívny priechod uzla je zabezpečený opäť funkciou `dump_node`, ktorá dostáva výsledok funkcie pristupovaný v uzle `FUNCTION_DECL` makrom `DECL_RESULT`.

Uzol `DECL_EXPR` bol obohatený o parameter `decl`, ktorý predstavuje deklaráciu premennej, ku ktorej sa daný deklaračný výraz vzťahuje. Keďže ide o hodnotu typu strom, na zaradenie do výstupu bola znova použitá funkcia `dump_node`.

Jednotlivé uzly deklarácií premenných sú v bloku prepojené jednosmerným spojovým zoznamom. Do výstupného súboru sa však táto informácia negenerovala. Pre uzol `VAR_DECL` som preto zaviedol parameter `chan`, kde hodnota tohto parametra predstavuje nasledujúcu deklaráciu premennej, do-

stupnú z aktuálnej deklarácie použitím makra `TREE_CHAIN`. Ak deklarácia premennej neobsahuje parameter `chan`, je poslednou v danom bloku.

Formát výstupu textových reťazcov bol upravený tak, aby boli všetky exportované reťazce začínajúce a ukončené znakom `"`. Vďaka tomu je možné bez problémov spracovávať reťazce znakov obsahujúce medzeru a akékoľvek iné znaky.

Patch súbory s popisom zmien sa nachádzajú na priloženom CD v adresári `src/patch`.

Príklad výstupu `original-raw` po úprave, pre program v jazyku C, obsahujúci deklaráciu funkcie `main` s prázdnym telom funkcie (program identický s referenčným príkladom `ex1_main.c`):

```
;; Function main (null)
;; enabled by -tree-original

@1    function_decl    name: @2     type: @3     srcp: "main.c":1
                       link: "extern"            body: @4
                       resd: @5
@2    identifier_node  strg: "main" lngt: 4
@3    function_type    unql: @6     size: @7     algn: 8
                       retn: @8
@4    statement_list   0   : @9     1   : @1
@5    result_decl      type: @8     scpe: @1     srcp: "main.c":1
                       note: "artificial"        size: @11
                       algn: 32
@6    function_type    size: @7     algn: 8      retn: @8
@7    integer_cst      type: @12    int : 8
@8    integer_type     name: @13    size: @11    algn: 32
                       prec: 32     sign: "signed"
                       min : @14    max : @15
@9    bind_expr        type: @16    body: @17
@10   return_expr      type: @16    expr: @18
@11   integer_cst      type: @12    int : 32
@12   integer_type     name: @19    size: @20    algn: 64
                       prec: 64     sign: "unsigned"
                       min : @21    max : @22
@13   type_decl        name: @23    type: @8
@14   integer_cst      type: @8     int : -2147483648
@15   integer_cst      type: @8     int : 2147483647
@16   void_type        name: @24    algn: 8
@17   statement_list
@18   modify_expr      type: @8     op 0: @5     op 1: @25
@19   identifier_node  strg: "bitsizetype"       lngt: 11
@20   integer_cst      type: @12    int : 64
```

```
@21   integer_cst      type: @12    int : 0
@22   integer_cst      type: @12    int : -1
@23   identifier_node  strg: "int"  lngt: 3
@24   type_decl        name: @26    type: @16
@25   integer_cst      type: @8     int : 0
@26   identifier_node  strg: "void" lngt: 4
```

3.1.1 Formátovanie zdrojového kódu v GCC

Pre úpravu zdrojových kódov je potrebné poznať zaužívané konvencie a dodržiavať ich. V zdrojových kódoch GCC platí, že zložené zátvorky vnoreného bloku (otváracia aj zatváracia), sú odsadené dvomi medzerami, s výnimkou najvyššej úrovne (napríklad deklarácia funkcie). V takom prípade sa zložené zátvorky neodsadzujú. Kód vo vnútri bloku je potom odsadený ďalšími dvoma medzerami. Osem po sebe idúcich medzier sa nahrádza tabulátorom. Názov funkcie, makra a pod., je od zátvoriek s parametrami oddelený medzerou.

```
int foo (int a, int b)
{
  if (a + b > 5)  // 2 medzery
    {             // 4 medzery
      if (a > 5)  // 6 medzier
        return a; // 1 tabulator
  ...
}
```

3.2 Integrácia prednej časti do GCC

Predná časť, ktorej realizácia je popísaná na nasledujúcich stranách, dostala pracovný názov Rawtree, odvodený od formátu výstupu (`original-raw`) a samotnej vnútornej reprezentácie, ktorú popisuje.

Zdrojové súbory prednej časti Rawtree sa nachádzajú v rovnomennom adresári `gcc/rawtree`. Ako základ implementácie bola použitá šablóna prednej časti [7]. Predná časť pozostáva z nasledujúcich súborov, požadovaných podľa [5]:

confg-lang.in
: Shell skript, ktorý definuje premenné, popisujúce jazyk. Tento súbor je povinný pre každú prednú časť.

Make-lang.in
: Súbor popisujúci závislosti pre zostavenie cieľových objektov. Je volaný v rámci `Makefile` vyššej úrovne.

Tabuľka 3.1: Tabuľka tokenov lexikálneho analyzátora

Token	Sekvencia	Význam
`RPTR`	`@2`	odkaz na iný uzol stromu
`COLON`	`:`	dvojbodka
`STR`	`abc`	reťazec bez medzier
`STR_RAW`	`"abc abc"`	reťazec začínajúci a ukončený znakom "
`NUMB`	`123`	číslo
`MINUS`	`-`	mínus

lang.opt
Obsahuje definície parametrov s ktorými môže byť prekladač spustený.

lang-specs.h
Záznam o prípustných koncovkách súborov daného jazyka. Jeho obsah je propagovaný do poľa `default_compilers` v súbore `gcc.c`.

***language*-tree.def**
Obsahuje definície nových kódov uzlov, ak jazyk nejaké zavádza. Tento súbor nie je povinný.

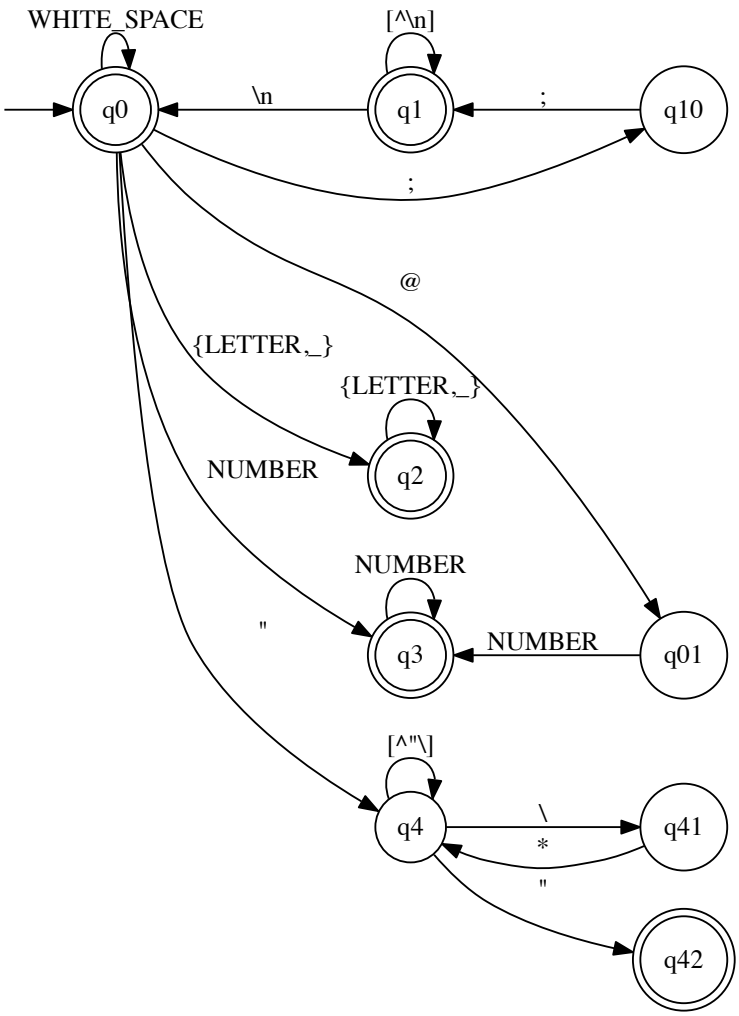
3.3 Lexikálna analýza

Úlohou lexikálnej analýzy je postupné čítanie znakov zo vstupu a ich transformácia na postupnosť *tokenov*, ktoré využíva syntaktická analýza. Na implementáciu lexikálneho analyzátora bol použitý konečný automat (obrázok 3.1), ktorý rozoznáva tokeny uvedené v tabuľke 3.1. Implementovaný lexikálny analyzátor vychádza z návrhu v [8, Kapitola 2.2]. Súbory, ktoré tvoria implementáciu lexikálneho analyzátora sa nachádzajú v adresári `rawtree/lexan`.

Samotný konečný automat je realizovaný Hardcoded prístupom, kedy sú jednotlivé stavy reprezentované miestom v programe (návestie) a prechody medzi nimi reprezentujú príkazy programu (`goto`) [8].

3.3.1 Detekcia chýb v lexikálnom analyzátore

V rámci lexikálnej analýzy sa prevádza základná kontrola vstupu. Na začiatok je prevedená kontrola existencie zadaného vstupného súboru. Ak súbor existuje, lexikálny analyzátor začne spracovávať vstup. Na vstupe sa môže objaviť neočakávaná sekvencia znakov, prípadne neočakávaný koniec súboru v komentári, alebo pri spracovávaní tokenu `STR_RAW`. Tieto chyby sú ošetrené volaním funkcie `error`, ktorej volajúci predáva chybovú hlášku ako C reťazec. Samotná funkcia potom k textu hlášky pridá informáciu o tom, na ktorom riadku došlo k uvedenej chybe.



Obr. 3.1: Konečný automat prevádzajúci lexikálnu analýzu vstupu

3.4 Syntaktická analýza

Syntaktická analýza je prevádzaná nad postupnosťou tokenov, prijatých z lekikálneho analyzátora. Na implementáciu bola použitá metóda *rekurzívneho zostupu*. Riadi sa $LL(1)$ gramatikou 2.5, ktorá generuje jazyk daný výstupným formátom GCC (`original-raw`). Súbory, ktoré tvoria implementáciu syntaktického analyzátora, sa nachádzajú v adresári `rawtree/parser`.

3.4.1 Priechody

Práca syntaktického analyzátora je rozdelená do niekoľkých priechodov.

Prvý priechod prevádza rekurzívny zostup, v rámci ktorého sa všetky informácie obsiahnuté vo vstupnom súbore zaznamenajú do vhodnej štruktúry vo forme spojových zoznamov. Každý uzol tak obsahuje spojový zoznam parametrov. Jednotlivé položky spojového zoznamu obsahujú názov parametra, hodnotu a odkaz na ďalší parameter, ak existuje.

Ďalší priechod zabezpečuje prípravu konkrétnych uzlov v poli, ktoré bude neskôr reprezentovať graf uzlov AST. V tomto priechode sa vytvárajú jednotlivé inštancie uzlov volaním príslušných konštruktorov, podľa typu uzla. V tomto bode je prevedená kontrola typu uzla. Ak typ uzla nie je podporovaný, prekladač sa ukončí. Zároveň sú v tomto priechode uložené všetky konštantné hodnoty členských premenných, volaním funkcie `addStatic`.

Po vytvorení všetkých uzlov je možné uložiť hodnoty parametrov, ktoré sú typu `RPTR`, teda sú odkazom na iné uzly. Týmto sa v poli `graph_nodes` vytvorí graf uzlov.

Posledný priechod prevádza samotný preklad abstraktného syntaktického stromu, do vnútornej reprezentácie GENERIC. Tento priechod sa prevádza nad grafom rekurzívne, prechádzaním grafu do hĺbky (bližšie popísané v kapitole 3.5.1).

Po preklade kompletnej deklarácie funkcie sa nad koreňovým uzlom volá funkcia `register_global_function_declaration`, ktorá zabezpečuje *gimplifikáciu* funkcie.

3.4.2 Detekcia chýb v syntaktickom analyzátore

Detekcia chýb prebieha predovšetkým na úrovni kontroly syntaxe. Okrem toho sa počas syntaktickej analýzy kontroluje postupnosť uzlov v rámci deklarácie funkcie. Uzly musia byť na vstupe označované od 1 po n, kde n je počet uzlov deklarovaných vo funkcii. Tieto uzly musia po sebe nasledovať v rámci jednej deklarácie funkcie vo vzostupnom poradí. Ak je táto podmienka porušená, prekladač ohlási chybu a ukončí sa.

3.5 Abstraktný syntaktický strom

Abstraktný syntaktický strom je reprezentovaný jednoduchou hierarchiou tried. Každý podporovaný typ uzla je reprezentovaný samostatnou triedou, pričom všetky sú zapúzdrené nadtriedou `CNode`. Definícia triedy `CNode` sa nachádza v súbore `gcc/rawtree/ast/ast.h`. Implementácia jednotlivých tried, reprezentujúcich podporované typy uzlov je rozdelená do 4 súborov (umiestnených v adresári `gcc/rawtree/ast`):

- decl.h
- expr.h
- type.h
- misc.h

3.5.1 Preklad AST do vnútornej reprezentácie GENERIC

Preklad uzlov grafu je realizovaný ako priechod grafom do hĺbky. Napriek tomu, že sa v rámci vnútorných reprezentácií najčastejšie používa výraz strom, ide o graf. Na konkrétny uzol totiž môže odkazovať viacero iných uzlov (napríklad uzol, reprezentujúci deklaráciu premennej, je len jeden, takže každá operácia s touto premennou bude odkazovať na jej deklaráciu), čo nie je v súlade s definíciou stromu, ktorý je definovaný ako súvislý graf bez kružníc, podľa [9]. Zavedenie dátovej štruktúry, kam budeme ukladať už preložené uzly, nám však umožní previesť DFS.

Na prácu s uzlami grafu som preto zaviedol pole `t_list`, ktorého položky sú typu `tree`. Na kontrolu, či je uzol preložený, som definoval makro `RT_TRANSLATE`, nasledovne:

```
#define RT_TRANSLATE(N) t_list[N->getN ()] \
? t_list[N->getN ()]  \
: N->translate ()
```

V prípade potreby akéhokoľvek uzla grafu nebudeme volať priamo metódu `translate`, ale použijeme vyššie uvedené makro

```
TREE_TYPE (tree) = RT_TRANSLATE (type);
```

Preklad uzlov je, podobne ako definícia tried AST, rozdelený do 4 súborov (umiestnených v adresári `gcc/rawtree/ast`):

- trans-decl.h
- trans-expr.h
- trans-type.h

- trans-misc.h

Nasledujúci popis jednotlivých uzlov vychádza z dokumentácie GCC [5], nie sú v ňom však zahrnuté všetky informácie o zmieňovaných uzloch. Ide predovšetkým o informácie, ktoré sú relevantné pre implementáciu uvedeného konceptu prednej časti. Zoznamy parametrov boli zostavené na základe testovania rôznych scenárov v programoch.

3.5.1.1 `FUNCTION_DECL`

Uzol `FUNCTION_DECL` reprezentuje deklaráciu funkcie. Predstavuje koreňový uzol stromu funkcie. Tento uzol je základom celého prekladu a preklad každej funkcie začína týmto uzlom. Obsahuje informácie o názve funkcie, jej parametre a telo a v neposlednom rade výsledok.

Prístup k názvu funkcie zabezpečuje makro `DECL_NAME`, a vracia nekvalifikovaný názov funkcie ako uzol `IDENTIFIER_NODE`. Pre prístup k názvu funkcie upravenému pre Assembler je možné použiť makro `DECL_ASSEMBLER_NAME`.

K parametrom funkcie je možné pristúpiť pomocou makra `DECL_ARGUMENTS`. Toto makro vracia prvý parameter funkcie. Ostatné parametre nasledujú v jednosmernom spojovom zozname a získať ich je možné postupne, volaním makra `TREE_CHAIN` na jednotlivé parametre.

Výsledok funkcie je dostupný použitím makra `DECL_RESULT`. Prístup k telu funkcie zabezpečuje makro `DECL_SAVED_TREE` (`body: @n` na výstupe), prípadne môže obsahovať `NULL_TREE`, pre funkciu bez tela (buď v kombinácii s nastavením `DECL_EXTERNAL` pre `body: undefined` alebo bez, pre úplné vynechanie položky `body` na výstupe). Makro `TREE_TYPE` bude v prípade deklarácie funkcie obsahovať uzol typu `FUNCTION_TYPE` alebo `METHOD_TYPE`. Položka stromu funkcie, pristupovaná makrom `DECL_INITIAL`, by mala byť nastavená, ak je funkcia definovaná v aktuálnej prekladovej jednotke. Zadná časť prekladača by však túto hodnotu nemala použiť.

`TREE_PUBLIC` obsahuje informáciu o tom, či je funkcia linkovaná externe (`link: extern`). `TREE_STATIC` obsahuje informáciu o tom, či bola daná funkcia definovaná.

3.5.1.2 `VAR_DECL`

Uzol `VAR_DECL` reprezentuje premennú s rozsahom jej platnosti. `DECL_SIZE` predstavuje uzol `IDENTIFIER_NODE` s názvom funkcie. `DECL_ALIGN` obsahuje informáciu o zarovnaní v bitoch, reprezentovanú ako `int`. `DECL_SIZE` predstavuje počet bitov, potrebných na reprezentáciu stromu a je reprezentovaný ako uzol typu `INTEGER_CST`. Predikáty `DECL_THIS_STATIC` a `DECL_THIS_EXTERN` hovoria o tom, či bola premenná deklarovaná ako statická a externá.

`DECL_INITIAL` obsahuje inicializačný výraz premennej. Výraz je vyhodnotený a jeho hodnota prekopírovaná do premennej, keď prekladač narazí na deklaračný výraz (uzol `DECL_EXPR`) danej premennej (pre globálne premenné

Tabuľka 3.2: Prehľad parametrov uzla `FUNCTION_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`name`	názov funkcie	Áno	`RPTR`
`mngl`	názov funkcie (Assembler)	Nie	`RPTR`
`type`	typ funkcie	Áno	`RPTR`
`srcp`	pozícia v zdrojovom súbore	Áno	`STR_RAW`
`link`	linkovanie funkcie	Áno	`STR_RAW`
`body`	telo funkcie	Nie	`RPTR`
`args`	parametre funkcie	Nie	`RPTR`
`resd`	návratová hodnota	Áno	`RPTR`

automaticky). Premenná existuje už pred spracovaním uzlu `DECL_EXPR` (v rozsahu jej platnosti), no jej hodnota je nedefinovaná.

V prípade, že po danej deklarácii premennej nasledujú ďalšie, bude táto obsahovať odkaz na nasledujúcu, dostupnú použitím makra `TREE_CHAIN`.

Tabuľka 3.3: Prehľad parametrov uzla `VAR_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`name`	názov premennej	Áno	`RPTR`
`type`	typ premennej	Áno	`RPTR`
`scpe`	rozsah platnosti	Nie	`RPTR`
`srcp`	pozícia v zdrojovom súbore	Áno	`STR_RAW`
`init`	inicializačný výraz	Nie	`RPTR`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`chan`	nasledujúca deklarácia	Nie	`RPTR`
`used`	použitie premennej	Áno	`NUMB`

3.5.1.3 `PARM_DECL`

Uzol `PARM_DECL` reprezentuje parameter funkcie. Prístup k jednotlivým parametrom tohto uzla je podobný, ako u uzla `VAR_DECL`. Makrom `DECL_ARG_TYPE` je možné pristupovať k typu, ktorý bude použitý pri odovzdávaní hodnoty funkcii.

3.5.1.4 `TYPE_DECL`

Tento uzol predstavuje deklaráciu typu (`typedef`). Makro `TREE_TYPE` obsahuje samotný uzol s typom, zatiaľ čo `TREE_NAME` obsahuje názov typu. V niektorých prípadoch deklarácia typu nemusí obsahovať názov.

Tabuľka 3.4: Prehľad parametrov uzla `PARM_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`name`	názov premennej	Áno	`RPTR`
`type`	typ premennej	Áno	`RPTR`
`scpe`	rozsah platnosti	Áno	`RPTR`
`srcp`	pozícia v zdrojovom súbore	Áno	`STR_RAW`
`argt`	typ paramtera	Áno	`RPTR`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`chan`	nasledujúca deklarácia	Nie	`RPTR`
`used`	použitie parametra	Áno	`NUMB`

Tabuľka 3.5: Prehľad parametrov uzla `TYPE_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`name`	názov typu	Nie	`RPTR`
`type`	typ	Áno	`RPTR`

3.5.1.5 `RESULT_DECL`

Uzol `RESULT_DECL` reprezentuje výsledok funkcie. Hodnota priradená tomuto uzlu predstavuje návratovú hodnotu funkcie, ktorá by mala byť vrátená volajúcemu. Podobne ako pri uzle `VAR_DECL`, aj tu je možné použiť makrá `DECL_SIZE` a `DECL_ALIGN`.

Tabuľka 3.6: Prehľad parametrov uzla `RESULT_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`scpe`	rozsah platnosti	Áno	`RPTR`
`srcp`	pozícia v zdrojovom súbore	Áno	`STR_RAW`
`note`	implicitná deklarácia	Nie	`STR_RAW`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`

3.5.1.6 `LABEL_DECL`

Uzol `LABEL_DECL` reprezentuje návestie v tele funkcie. Makro `DECL_CONTEXT` obsahuje informáciu o rozsahu platnosti návestia (deklarácia funkcie).

Tabuľka 3.7: Prehľad parametrov uzla `LABEL_DECL`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`scpe`	rozsah platnosti	Áno	`RPTR`
`note`	implicitná deklarácia	Nie	`STR_RAW`

3.5.1.7 `VOID_TYPE`

Tento uzol reprezentuje typ `void`.

Na reprezentáciu uzlu `VOID_TYPE` bola použitá premenná `void_type_node`, ktorá je vstavanou premennou a reprezentuje uzol tohto typu.

Tabuľka 3.8: Prehľad parametrov uzla `VOID_TYPE`

Parameter	Význam	Povinný	Typ hodnoty
`name`	názov	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`

3.5.1.8 `INTEGER_TYPE`

Uzol `INTEGER_TYPE` reprezentuje celočíselné typy (napr. `char`, `int`, `long` ...). Makro `TYPE_PRECISION` vracia počet bitov použitých na reprezentáciu (reprezentované ako `unsigned int`). `TYPE_PRECISION` a `TYPE_SIZE` vo všeobecnosti nevyjadrujú to isté. Zatiaľ čo `TYPE_PRECISION` predstavuje počet bitov potrebných pre reprezentáciu typu, `TYPE_SIZE` je počet bitov na reprezentáciu po zarovnaní. Celočíselný typ je bezznamienkový, ak je nastavený flag `TYPE_UNSIGNED`, inak ide o znamienkový typ. `TYPE_MIN_VALUE` predstavuje najmenšiu hodnotu, ktorá môže byť reprezentovaná daným typom. Je reprezentovaná uzlom `INTEGER_CST`. Obdobne, celočíselné typy obsahujú maximálnu hodnotu, ktorú je možné reprezentovať daným typom, prístupnú makrom `TYPE_MAX_VALUE`.

3.5.1.9 `FUNCTION_TYPE`

Uzol `FUNCTION_TYPE` reprezentuje typ funkcie, ktorá nie je členskou metódou, prípadne je statickou členskou metódou (na reprezentáciu členských metód, ktoré nie sú statické, sa používa uzol `METHOD_TYPE`, ktorý nie je v implementovanej prednej časti podporovaný).

Makro `TREE_TYPE` vracia návratový typ funkcie. `TYPE_ARG_TYPES` je zoznam typov parametrov, reprezentovaný ako uzol `TREE_LIST`.

Tabuľka 3.9: Prehľad parametrov uzla `INTEGER_TYPE`

Parameter	Význam	Povinný	Typ hodnoty
`qual`	kvalifikátory	Nie	`STR`
`name`	názov	Nie	`RPTR`
`unql`	nekvalifikovaný typ	Nie	`RPTR`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`prec`	presnosť	Áno	`NUMB`
`sign`	znamienko	Áno	`STR_RAW`
`min`	minimálna hodnota	Áno	`RPTR`
`max`	maximálna hodnota	Áno	`RPTR`

Tabuľka 3.10: Prehľad parametrov uzla `FUNCTION_TYPE`

Parameter	Význam	Povinný	Typ hodnoty
`unql`	nekvalifikovaný typ	Nie	`RPTR`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`retn`	návratový typ	Áno	`RPTR`
`prms`	parametre	Nie	`RPTR`

3.5.1.10 `POINTER_TYPE`

Uzol `POINTER_TYPE` reprezentuje ukazovateľové typy. Na prístup k typu, na ktorý ukazuje slúži makro `TREE_TYPE`.

Tabuľka 3.11: Prehľad parametrov uzla `POINTER_TYPE`

Parameter	Význam	Povinný	Typ hodnoty
`qual`	kvalifikátory	Nie	`STR`
`unql`	nekvalifikovaný typ	Nie	`RPTR`
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`ptd`	typ na ktorý ukazuje	Áno	`RPTR`

3.5.1.11 `ARRAY_TYPE`

Uzol `ARRAY_TYPE` reprezentuje typ pole. Makrom `TREE_TYPE` sa pristupuje k typu elementov poľa. Ak má pole nastavené medze, `TYPE_DOMAIN` obsahuje uzol `INTEGER_TYPE`, ktorého uzly `TYPE_MIN_VALUE` a `TYPE_MAX_VALUE` predstavujú dolnú a hornú medzu poľa.

Tabuľka 3.12: Prehľad parametrov uzla `ARRAY_TYPE`

Parameter	Význam	Povinný	Typ hodnoty
`size`	veľkosť (po zarovnaní)	Áno	`RPTR`
`algn`	zarovnanie	Áno	`NUMB`
`elts`	typ elementov poľa	Áno	`RPTR`
`domn`	uzol obsahujúci medze poľa	Nie	`RPTR`

3.5.1.12 `BIND_EXPR`

Uzol `BIND_EXPR` reprezentuje prepojenie bloku s jeho lokálnymi premennými. Ide o jeden zo základných uzlov, ktorý sa napr. vyskytuje v každom strome deklarácie funkcie (aj vo funkciách s prázdnym telom).

Premenné, deklarované v bloku, sú dostupné makrom `BIND_EXPR_VARS`. Premenné po sebe nasledujú v jednosmernom spojovom zozname v poradí, v akom boli deklarované. Ďalšiu premennú získame makrom `TREE_CHAIN`. Samotné telo bloku je dostupné makrom `BIND_EXPR_BODY`.

Tabuľka 3.13: Prehľad parametrov uzla `BIND_EXPR`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`vars`	deklarácie premenných	Nie	`RPTR`
`body`	telo (zoznam príkazov)	Áno	`RPTR`

3.5.1.13 `DECL_EXPR`

Uzol `DECL_EXPR` reprezentuje lokálnu deklaráciu. Samotná entita, ktorú daný výraz deklaruje, je dostupná makrom `DECL_EXPR_DECL`.

Tabuľka 3.14: Prehľad parametrov uzla `DECL_EXPR`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`decl`	deklarácia entity	Áno	`RPTR`

3.5.1.14 `RETURN_EXPR`

Uzol `RETURN_EXPR` reprezentuje príkaz `return` vo funkcii. Jeho operandom je návratová hodnota (dostupná makrom `TREE_OPERAND (node, 0)`). Operand tohto uzlu bude obsahovať hodnotu `NULL_TREE`, pre príkaz `return;` (bez návratovej hodnoty).

Tabuľka 3.15: Prehľad parametrov uzla `RETURN_EXPR`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`expr`	operand 0	Nie	`RPTR`

3.5.1.15 `CALL_EXPR`

Uzol `CALL_EXPR` reprezentuje volanie funkcie. Ukazovateľ na volanú funkciu je možné získať makrom `CALL_EXPR_FN`. K jednotlivým parametrom je možné pristupovať makrom `CALL_EXPR_ARG`. V prípade potreby je možné iterovať parametrami makrom `FOR_EACH_CALL_EXPR_ARG`.

Tabuľka 3.16: Prehľad parametrov uzla `CALL_EXPR`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`fn`	ukazovateľ na volanú funkciu	Áno	`RPTR`
n	n-tý parameter volania	Nie	`RPTR`

3.5.1.16 `ADDR_EXPR`

Uzol `ADDR_EXPR` reprezentuje adresu objektu. Typ tohto výrazu môže byť `POINTER_TYPE`, alebo `REFERENCE_TYPE`. Jeho operandom môže byť deklarácia, alebo iný výraz.

Tabuľka 3.17: Prehľad parametrov uzla `ADDR_EXPR`

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`op 0`	operand 0	Áno	`RPTR`

3.5.1.17 `NOP_EXPR`

Uzol `NOP_EXPR` reprezentuje konverziu, ktorá nevyžaduje generovanie kódu (napríklad konverzia z `char*` na `int*`). Operandom je výraz, ktorý má byť konvertovaný.

3.5.1.18 `GOTO_EXPR`

Uzol `GOTO_EXPR` reprezentuje príkaz `goto`. Cieľ skoku je možné získať použitím makra `GOTO_DESTINATION`.

Tabuľka 3.18: Prehľad parametrov uzla NOP_EXPR

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
op 0	operand 0	Áno	RPTR

Tabuľka 3.19: Prehľad parametrov uzla GOTO_EXPR

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
labl	cieľové návestie	Áno	RPTR

3.5.1.19 LABEL_EXPR

Uzol LABEL_EXPR reprezentuje návestie. Samotnú deklaráciu návestia je možné z tohto uzla získať použitím makra LABEL_EXPR_LABEL.

Tabuľka 3.20: Prehľad parametrov uzla LABEL_EXPR

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
name	názov návestia	Áno	RPTR

3.5.1.20 COND_EXPR

Uzol COND_EXPR reprezentuje podmienku (? :). Operand 0 je celočíselného alebo búlovského typu. Ak sa vyhodnotí ako nenulový, vyhodnotí sa operand 1, ktorý je zároveň vrátený ako hodnota výrazu. Ak je operand 0 vyhodnotený ako nulový, vyhodnotí sa operand 2 a jeho hodnota je vrátená ako hodnota výrazu. Doležité je tomuto uzlu nastaviť flag TREE_SIDE_EFFECTS, aby sa výraz vyhodnotil.

Tabuľka 3.21: Prehľad parametrov uzla COND_EXPR

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
op 0	operand 0	Áno	RPTR
op 1	operand 1	Áno	RPTR
op 2	operand 2	Áno	RPTR

3.5.1.21 ARRAY_REF

Uzol ARRAY_REF reprezentuje prístup do poľa. Operand 0 je pole, do ktorého sa bude pristupovať. Operand 1 predstavuje pristupovaný index. Uzol obsahuje ďalšie dva operandy, kde operand 2 predstavuje začiatok poľa a operand 3 je veľkosť prvku, tieto sa však nastavujú až v procese gimplifikácie, a teda ich nie je potrebné nastavovať. Na vytvorenie tohto uzla je však preto nutné použiť funkciu build4, kde posledné dva parametre budú NULL_TREE.

Tabuľka 3.22: Prehľad parametrov uzla ARRAY_REF

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
op 0	operand 0	Áno	RPTR
op 1	operand 1	Áno	RPTR

3.5.1.22 Výrazy s dvomi operandmi

Do skupiny výrazov s dvomi operandmi patria uzly:

MODIFY_EXPR
COMPOUND_EXPR
PLUS_EXPR
MINUS_EXPR
MULT_EXPR
TRUNC_DIV_EXPR
TRUNC_MOD_EXPR
POINTER_PLUS_EXPR
TRUTH_AND_EXPR
TRUTH_OR_EXPR
TRUTH_XOR_EXPR
TRUTH_ORIF_EXPR
TRUTH_ANDIF_EXPR
BIT_IOR_EXPR
BIT_XOR_EXPR
BIT_AND_EXPR
POSTINCREMENT_EXPR
POSTDECREMENT_EXPR
LT_EXPR
LE_EXPR
GT_EXPR
GE_EXPR
EQ_EXPR
NE_EXPR

Všetky vyššie uvedené uzly sa prekladajú rovnako. Typ uzla je možné získať použitím makra `TREE_TYPE`. Na prístup k jednotlivým operandom potom slúži makro `TREE_OPERAND (node, n)`, kde `node` je daný uzol a `n` je poradové číslo operandu (operandy sú číslované od nuly).

Pri tvorbe prednej časti som sa rozhodol reprezentovať každý z týchto uzlov jedinečnou triedou. Na jednej strane je toto riešenie pomerne neefektívne, predovšetkým z hľadiska veľkosti kódu, cieľom však bolo zachovať jednoduchosť a priamočiarosť riešenia. Dôraz bol kladený predovšetkým na funkčnosť a rozšíriteľnosť, keďže implementovaná bola len časť existujúcich uzlov. V budúcnosti by sa však triedy reprezentujúce dvojoperandové výrazy s principiálne rovnakým prekladom mohli zlúčiť do jednej triedy, kde by navyše pribudla informácia o kóde uzlu.

Tabuľka 3.23: Prehľad parametrov uzlov s dvomi operandmi

Parameter	Význam	Povinný	Typ hodnoty
`type`	typ	Áno	`RPTR`
`op 0`	operand 0	Áno	`RPTR`
`op 1`	operand 1	Áno	`RPTR`

3.5.1.23 `STATEMENT_LIST`

Uzol `STATEMENT_LIST` reprezentuje postupnosť príkazov na rovnakej úrovni programu. Na vytvorenie nového uzla typu `STATEMENT_LIST` slúži funkcia `alloc_stmt_list`. Následné pridanie príkazu do zoznamu je možné vykonať pomocou funkcie `append_to_statement_list`. Ďalšie funkcie pre manipuláciu s uzlom `STATEMENT_LIST` sú uvedené v súbore `gcc/tree-iterator.h`.

Tabuľka 3.24: Prehľad parametrov uzla `STATEMENT_LIST`

Parameter	Význam	Povinný	Typ hodnoty
n	n-tý príkaz na danej úrovni	Nie	`RPTR`

3.5.1.24 `IDENTIFIER_NODE`

Uzol `IDENTIFIER_NODE` reprezentuje identifikátor. Hodnota parametra, reprezentujúca reťazec, bola pôvodne typu `STR`, v rámci generovania výstupu bol však formát tohto parametru upravený kvôli identifikátorom s medzerou (napríklad `unsigned int`). Po úprave sa hodnota generuje ako `STR_RAW`. Pre prístup k samotnému reťazcu znakov, tvoriacich identifikátor, slúži makro `IDENTIFIER_POINTER`. Reťazec je reprezentovaný ako `char*`, a je vždy ukončený znakom `NUL`. Makrom `IDENTIFIER_LENGTH` je možné pristúpiť k dĺžke

identifikátora, do ktorej sa nepočíta ukončovací znak NUL. Vrátená hodnota je vždy rovnaká ako strlen (IDENTIFIER_POINTER (t)).

Tabuľka 3.25: Prehľad parametrov uzla IDENTIFIER_NODE

Parameter	Význam	Povinný	Typ hodnoty
strg	reťazec znakov	Áno	STR_RAW
lngt	dĺžka reťazca	Áno	NUMB

3.5.1.25 INTEGER_CST

Uzol INTEGER_CST reprezentuje celočíselnú konštantu. Jej typ je dostupný použitím makra TREE_TYPE. Na vytváranie uzlov celočíselných konštánt som v implementácii použil vstavanú funkciu build_int_cst.

Tabuľka 3.26: Prehľad parametrov uzla INTEGER_CST

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
int	celočíselná konštanta	Áno	NUMB

3.5.1.26 STRING_CST

Uzol STRING_CST reprezentuje konštantu - reťazec znakov. Na prístup k dĺžke reťazca slúži makro TREE_STRING_LENGTH. Samotný reťazec je dostupný makrom TREE_STRING_POINTER, ako char*. Reťazec nemusí byť ukončený znakom NUL, preto TREE_STRING_LENGTH zahŕňa aj ukončovací znak NUL.

Tabuľka 3.27: Prehľad parametrov uzla STRING_CST

Parameter	Význam	Povinný	Typ hodnoty
type	typ	Áno	RPTR
strg	reťazec znakov	Áno	STR_RAW
lngt	dĺžka reťazca	Áno	NUMB

3.5.1.27 TREE_LIST

Uzol TREE_LIST reprezentuje jednu z dvoch základných kontajnerových dátových štruktúr, ktoré sa dajú reprezentovať priamo uzlami stromu. TREE_LIST je jednosmerný spojový zoznam, ktorý obsahuje dva stromy v jednom uzle. TREE_VALUE nesie základné informácie, ktoré môžu byť doplnené druhým uzlom, ktorý je dostupný ako TREE_PURPOSE. K odkazu na ďalšiu položku zoznamu sa potom pristupuje pomocou makra TREE_CHAIN.

Tabuľka 3.28: Prehľad parametrov uzla TREE_LIST

Parameter	Význam	Povinný	Typ hodnoty
valu	hodnota	Áno	RPTR
purp	tag (doplňujúce informácie)	Nie	RPTR
chan	nasledujúca položka zoznamu	Nie	RPTR

3.6 Výstup pre spracovanie programom dot

Motiváciou pre generovanie obrazového výstupu vnútornej reprezentácie programu v GCC je predovšetkým lepšia a jednoduchšia predstava o tom, ako je program vo vnútri GCC reprezentovaný. Textový výstup je v podstate len pretransformovaný formát spracovávaného vstavaného výstupu original-raw do formátu vhodného na spracovanie programom dot, ktorý z neho dokáže vygenerovať grafický výstup vo forme grafu. Na základe takto vytvoreného grafu je potom omnoho jednoduchšie hľadať a odstraňovať prípadné chyby.

Na ovládanie generovania súborov vo formáte dot pre jednotlivé funkcie bol v rámci prednej časti implementovaný prepínač -print.

Generovanie výstupu je rozdelené do 4 súborov (umiestnených v adresári gcc/rawtree/ast):

- print-decl.h
- print-expr.h
- print-type.h
- print-misc.h

Na vygenerovanie obrazového výstupu vo zvolenom formáte je potrebné mať nainštalovaný program Graphviz, pretože na preklad potrebujeme jeho filter dot. Ten je primárne určený na vytváranie orientovaných grafov. Jeho základné použitie je jednoduché:

```
dot -Tformat input_file -o output_file
```

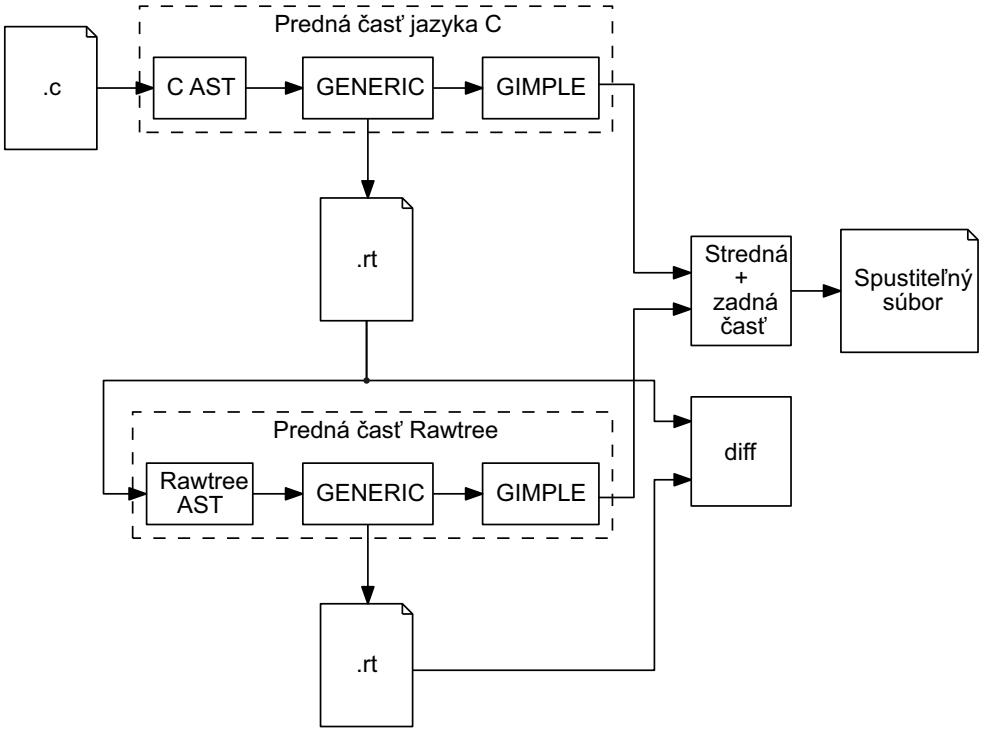
Výsledné grafy sú dostačujúce pre vytvorenie lepšej predstavy o vnútornej štruktúre programu. Časť informácií však musela byť odstránená kvôli prehľadnosti, takže statické parametre obsahujú len niektoré uzly, u ktorých sú tieto informácie zásadné. Väčšina uzlov však obsahuje len odkazy na svoje parametre. Ukážka grafického výstupu programu dot z vygenerovaného výstupu sa nachádza v prílohe A.1.

Najväčším problémom zobrazovania stromov funkcií je skutočnosť, že už pre miniatúrne funkcie obsahujú ich grafy veľké množstvo uzlov, a ešte väčšie množstvo hrán medzi nimi. Grafy sú preto generované pre každú funkciu zvlášť (do iného súboru).

Kapitola 4

Testovanie

Na účely testovania bola použitá jednoduchá schéma, ktorá umožňuje striktné overenie prekladu, porovnaním výstupu prednej časti, spracovávajúcej vstupný jazyk a výstupu prednej časti Rawtree, ako na obrázku 4.1.



Obr. 4.1: Schéma testovania prednej časti

Najskôr je potrebné preložiť program z jazyka C a vygenerovať pri jeho preklade výstup vo formáte `original-raw`. Výsledkom takéhoto prekladu sú dva súbory - spustiteľný program a výstup vnútornej reprezentácie. Následne je potrebné preložiť vyššie uvedeným postupom získaný výstup vnútornej re-

prezentácie pomocou prednej časti Rawtree. Pri preklade je potrebné opäť vygenerovať výstup vnútornej reprezentácie a spustiteľný program. Posledným krokom testovacieho procesu je porovnanie výstupov (striktne, napr. pomocou programu `diff`) a funkcionality spustiteľných programov.

Súbor s príponou `.rt` je možné preložiť z adresára `gcc-obj/gcc`, nasledovne:

```
./xrt -fdump-tree-original=dump.rt -B . input_file.rt
```

4.1 Testovacie programy

Na testovanie som pripravil 20 programov v jazyku C, ktoré sa nachádzajú v adresári `gcc/rawtree/testsuite/reference`. Ide prevažne o jednoduché programy, ktoré ukazujú funkčnosť konceptu na základných jazykových konštrukciách. Prvý program obsahuje deklaráciu funkcie `main` bez tela. V ďalších príkladoch je postupne demonštrovaná podpora rôznych typov uzlov. V poslednom programe je spolu s funkciou `main` implementovaných 9 rôznych funkcií, použitých v iných príkladoch.

4.2 Testovací skript

Na zjednodušenie testovania bol zavedený testovací skript `test`, nachádzajúci sa v adresári `gcc/rawtree/testsuite`, ktorý umožňuje preklad a porovnanie výstupov všetkých referenčných programov naraz. V tomto adresári sa nachádza podadresár `reference`, ktorý obsahuje testovacie programy v jazyku C, výstup `original-raw` pre každý z nich, rovnako ako spustiteľný súbor pre každý z týchto programov. V podadresári `output` sa nachádzajú súbory vygenerované prednou časťou (`original-raw` a spustiteľné súbory). Testovací skript vykonáva testovanie pre každý z prílkadov tak, ako je uvedené na obrázku 4.1.

4.3 Výsledky testovania

Testovanie prebehlo úspešne na všetkých pripravených programoch. Výstup, vygenerovaný pri preklade prednou časťou Rawtree je totožný so vstupom a funkcionalita spustiteľných súborov tiež zodpovedá referenčnej. Jediný problém na ktorý som počas testovania narazil je problém s poliami, ktorý sa nepodarilo opraviť. Z doteraz neznámych príčin práca so staticky alokovaným poľom celočíselného typu, ktoré obsahuje viac ako 4 položky, spôsobí chybu segmentácie pamäti. Aj v tomto prípade však výstup zodpovedá referenčnému výstupu.

Záver

Výsledkom práce je funkčný koncept prednej časti prekladača, ktorá je schopná prekladať výstup vnútornej reprezentácie GCC vo formáte `original-raw`, do vnútornej reprezentácie GENERIC, pre základnú podmnožinu typov uzlov GENERIC.

Práca bola zameraná predovšetkým na overenie konceptu. Otázkou spočiatku bolo, či bude možné program na základe výstupu rekonštruovať. Počas práce sa ukázalo, že vstavaný výstup formátu `original-raw` poskytuje o programe veľmi detailné informácie, napriek tomu časť informácii potrebných k rekonštrukcii chýbala.

Úprava formátu neskôr umožnila získanie všetkých potrebných informácii o programe, a tým úspešný preklad do vnútornej reprezentácie GENERIC.

V budúcnosti by som sa chcel venovať rozšíreniu podpory uzlov a v prípade úspešného dokončenia by som rád integroval túto prednú časť do oficiálneho vydania GCC. Aktuálna implementácia tiež poskytuje mnoho priestoru na zlepšovanie výkonových parametrov, keďže hlavným cieľom tejto práce bolo vytvorenie funkčného konceptu.

Literatúra

[1] GCC Development Plan. 2001, [cit. 2015-01-02]. Dostupné z: `https://gcc.gnu.org/develop.html`

[2] A Brief History of GCC. 2008, [cit. 2015-01-03]. Dostupné z: `https://gcc.gnu.org/wiki/History`

[3] GCC mirror sites. [cit. 2015-01-02]. Dostupné z: `https://gcc.gnu.org/mirrors.html`

[4] Installing GCC: Configuration. [cit. 2015-01-05]. Dostupné z: `https://gcc.gnu.org/install/configure.html`

[5] Stallman, R.; the GCC Developer Community: *GNU Compiler Collection Internals*. 2014, [cit. 2015-01-10]. Dostupné z: `https://gcc.gnu.org/onlinedocs/gccint/`

[6] Debugging options. [cit. 2015-01-08]. Dostupné z: `https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`

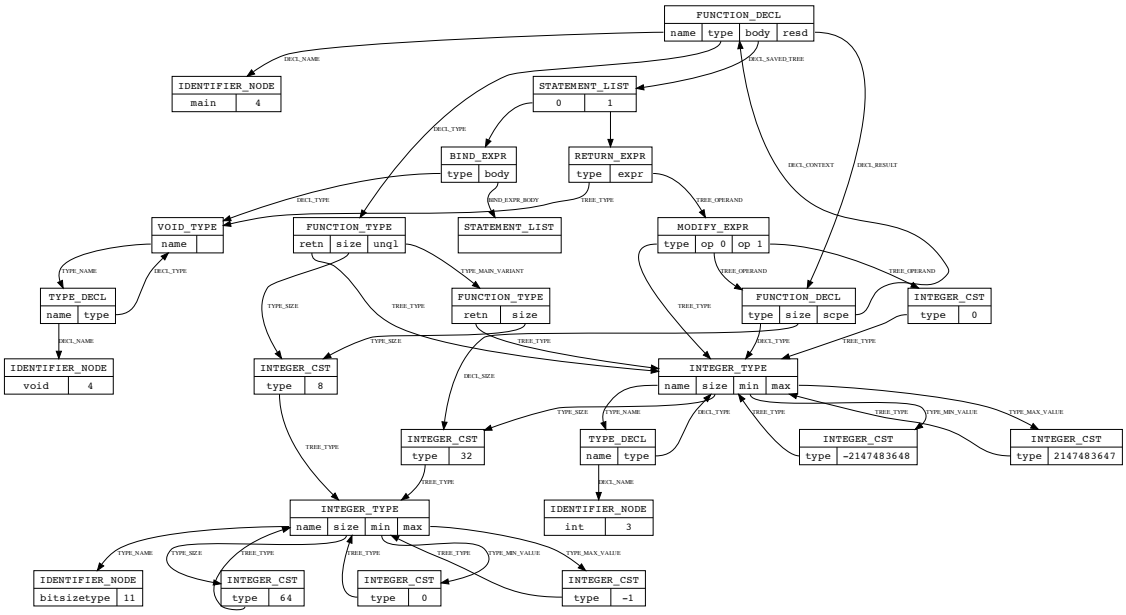
[7] Trávníček, J.: Sample front-end. 2014. Dostupné z: `http://git.extremehost.cz/gitweb/`

[8] Müller, K.: *Programovací jazyky*. České vysoké učení technické v Praze, 2001, ISBN 80-01-02458-X.

[9] Kolář, J.: *Teoretická informatika*. Česká informatická společnost, 2000, ISBN 9788090085381.

Dodatok **A**

Obrazová príloha



Obr. A.1: Ukážka výstupu programu dot (funkcia `main`)

Dodatok B

Zoznam použitých skratiek

GCC GNU Compiler Collection

FSF Free Software Foundation

EGCS Experimental/Enhanced GNU Compiler System

RTL Register Transfer Language

AST Abstract Syntax Tree

DFS Depth-First Search

SVN Subversion

Dodatok C

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `rawtree` ..... implementácia prednej časti prekladača
  - `patch` ..... adresár s vytvorenými záplatami
  - `tex` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `BP_Senko_Martin_2015.pdf` ..... text práce vo formáte PDF