Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Porovnání implementací předních částí překladačů GCC a LLVM

Ivan Ryšavý

Vedoucí práce: Ing. Jan Trávníček

27. dubna 2015

Poděkování

Děkuji své přítelkyni Klárce za podporu a pochopení. Děkuji svému vedoucímu práce Ing. Janu Trávníčkovi za podnětné připomínky. Nakonec děkuji i svému věrnému příteli Vincentovi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 27. dubna 2015 .....

Odkaz na tuto práci

Ryšavý, Ivan. *Porovnání implementací předních částí překladačů GCC a LLVM.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Práce se zabývá implementací předních částí překladačů GCC a LLVM. Přední části zpracují a analyzují zdrojový kód jazyka Mila, který transformují ve vnitřní formu daného překladače. Jazyk Mila je zde rozebrán a je pro něj implementován lexikální analyzátor, parser a abstraktní syntaktický strom, což je v práci dokumentováno.

Vpráci je kladen důraz na detailní popis a porovnání využitých rozhraní GCC a LLVM. Na základě tohoto popisu může kdokoliv další vytvořit novou přední část pro zmíněné překladače. Porovnáním bylo zjištěno, že implementace přední části pro LLVM je oproti GCC náročnější, protože se od přední části vyžaduje větší předzpracování zdrojového kódu.

Obě přední části byly úspěšně implementovány a jsou otestovány sadou vzorových programů, které pokrývají všechny implementované funkcionality. Zdrojové kódy předních částí a vzorových programů jsou spolu s kompletní LL(k) gramatikou jazyka Mila k nalezení v příloze.

Klíčová slova GCC, LLVM, Přední část překladače, Rozhraní překladače, Lexikální analýza, Syntaktická analýza, Abstraktní syntaktický strom, Jazyk Mila

Abstract

Implementations of compiler frontends for GCC and LLVM compiler toolkits are presented in the thesis. Frontends parse and analyse Mila source code, transforming it into an intermediate representation used by a respective compiler. Mila language is formally described and an implementation of a lexical analyser, a syntactic analyser and an abstract syntax tree is documented.

In this thesis there is placed emphasis on the detailed description and comparison of used interface of GCC and LLVM. Based on this description, anyone can create a new frontend for these compilers. It was found that the implementation of the LLVM front-end is more difficult, because there is needed to do more transformations of source code in the front-end.

Front-ends have been successfully implemented and tested to be correct by a set of sample programs covering all functionality. Source codes of front-ends, sample programs and complete LL(k) grammar of Mila language can be found in the appendix.

Keywords GCC, LLVM, Compiler frontend, Compiler interface, Lexical analysis, Syntactic analysis, Abstract syntax tree, Mila language

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je psán software v naprosté většině případů pomocí programovacích jazyků vyšší úrovně. Výhoda těchto programovacích jazyků je nepopiratelná - programy v nich lze pohodlně vytvářet i upravovat. Výsledné zdrojové kódy softwarového díla ale musí být před spuštěním nejprve přeloženy překladačem. Překladač se tak stává důležitým nástrojem pro všechny programátory.

Pokud je vymyšlen nový programovací jazyk, tak je tedy potřeba k němu vyrobit překladač. Zároveň ale platí, že překladač není třeba stavět na zelené louce.

Tato práce se zabývá tvorbou překladače způsobem, při němž se využije návrhu existujících překladačů, které se skládají typicky z několika částí, a implementuje se pouze ta část, která je odpovědná za zpracování jazyka. Pokud se pak nově implementovaná část spojí se zbytkem, bude vzešlý překladač maximálně těžit z existující infrastruktury a generovat spustitelný kód pro všechny podporované cílové platformy.

Dílo navazuje na semestrální práci psanou v rámci předmětu programovací jazyky a překladače. V rámci semestrální práce byla vyvinuta část překladače pro GCC, která je zde dále rozvinuta. Přínos bakalářské práce je však především v prozkoumání možností LLVM a sepsání procesu tvorby, včetně popisu jednotlivých rozhraní. Práce tak může sloužit jako návod pro konstrukci předních částí překladačů GCC a LLVM.

Kapitola 1

Cíl práce

První z cílů práce je implementace předních částí pro GCC a LLVM, které musí být schopny zpracovat zdrojový kód v zadaném jazyku Mila. V práci bude rozebrána architektura předních částí a následně bude vysvětleno, jak probíhá zpracování zdrojového kódu.

Kromě implementace dvou předních částí pro jazyk Mila zde budou ještě vytvořeny dvě přední části, které nezpracovávají žádný vstupní jazyk, ale pouze generují minimální spustitelný program. Tyto minimální přední části tak mohou sloužit jako šablona pro tvorbu nové přední části.

Hlavní cíl práce je však na základě implementací popsat jednotlivá využitá rozhraní GCC a LLVM a obě tato rozhraní porovnat. Smyslem je vytvořit ucelený popis, který bude moci sloužit jako návod pro kohokoliv, kdo bude chtít implementovat svou vlastní přední část překladače.

Pro přední části bude v rámci práce nakonec vytvořena sada testovacích programů, kterými bude ověřena korektnost implementace.

Kapitola 2

Analýza a návrh

2.1 Překladač

Překlad z formy psané programátorem do cílového spustitelného souboru je možno provádět mnoha způsoby. V nejjednodušším případě píše programátor rovnou strojový kód (např. pomocí hexa editoru) a překladač tak není vůbec potřeba. Psaní strojového kódu však není příliš efektivní, kvůli nutnosti psát pro každou platformu kód znovu. Další nevýhodou je nízkoúrovňovost daného řešení, jelikož i jednoduché programy jsou ve strojovém kódu poměrně rozsáhlé, obtížně čitelné a jejich změna může stát mnoho času [13].

Jako vyšší formu by bylo možno považovat jazyk symbolických adres. JSA umožňuje psát vykonávané instrukce dané platformy v textové podobě a podporuje mimo jiné návěstí, takže odpadá nutnost psát kód s absolutními skoky. Tento jazyk je sice stále nízkoúrovňový, ale už je pro člověka lépe čitelný a změny se v něm dělají lépe. Nutnost psát programy pro každou platformu znovu však neodpadá. Aby šlo spustit program zapsaný v JSA, tak je potřeba překladač nebo interpret. Interpretem se tato práce zabývat nebude. Překladač na vstupu dostane text, který zpracovává, a na výstupu vytváří výsledný strojový kód. Vzhledem k nízkoúrovňovosti mohou být takovéto překladače relativně jednoduché a kromě parsování textu se od nich např. předpokládá již zmíněné přepočítání symbolických adres [13].

Aby bylo dosaženo rozumné efektivity psaní kódu, čitelnosti výsledného programu a přenositelnosti mezi platformami, byly vymyšleny programovací jazyky vyšší úrovně. Tyto jazyky se už neskládají z instrukcí dané platformy, ale dovolují programátorovi psát programy pomocí vyšších konstrukcí. Umožňují deklarovat funkce a proměnné, kromě skoků podporují í podmíněné příkazy a cykly, a abstrahují tak cílovou platformu. U některých těchto jazyků může být programátorovi ve výsledku jedno na jaké cílové platformě jeho kód poběží. Z těchto specifikací plyne, že překladač, který takovýto jazyk překládá, už musí být mnohem složitější než překladač JSA [15].

Překladače vyšších programovacích jazyků se díky zmíněným vlastnostem skládají typicky z několika částí. Je zde přední část, která je zodpovědná za přečtení vstupu od programátora a zadní část, která je zodpovědná za vygenerování spustitelného kódu pro danou platformu. Mezi přední a zadní částí může být ještě prostřední část, která abstrahuje zadní část a provádí se zde například strojově nezávislé optimalizace [2].

2.2 Jazyk Mila

Mila je imperativní programovací jazyk inspirovaný jazykem Pascal. Jazyk umí deklarovat funkce, proměnné typu celé číslo, pole fixní velkosti, aritmeticko-logické výrazy, volání funkcí (i v rekurzi), podmíněné příkazy, for a while cykly, načítat data ze standardního vstupu a psát na standardní výstup.

Pro názornost je zde uveden příklad zdrojového kódu 2.1 v jazyku Mila. Ukázkový program počítá největšího společného dělitele. Zdrojový kód obsahuje rekurzivní funkci gcd, která provádí samotný výpočet. V hlavním bloku programu jsou deklarovány proměnné a a b, které jsou nejprve přečteny ze standardního vstupu a následně jako argumetny předány funkci gcd, která je zavolána. Návratová hodnota volání funkce gcd je nakonec vypsána na standardní výstup.

Formálně lze jazyk popsat například rozvinutou Backus-Naurovo formou, která je uvedena v tabulce 2.1

```
1  program gcd;
2
3  function gcd(a: integer; b: integer): integer;
4  begin
5    if b = 0 then
6      gcd := a
7    else
8      gcd := gcd(b, a~mod b);
9  end;
10
11 var a, b: integer;
12 begin
13   readln(a);
14   readln(b);
15   writeln(gcd(a, b));
16 end.
```

Zdrojový kód 2.1: Ukázka programu v jazyce Mila

Tabulka 2.1: Jazyk Mila - EBNF

Program	=	Header { FunctionDeclaration ';' } Block '.'
Header	=	'program' IDENT ';'
FunctionDeclaration	=	'procedure' IDENT '(' ParameterList ')' ';' Block
		| 'function' IDENT '(' ParameterList ')' ':' VarType ';' Block
ParameterList	=	Parameter { ',' Parameter } | ε
Parameter	=	IDENT ':' VarType
Block	=	{ ConstDeclaration } { VarDeclaration } InnerBlock
ConstDeclaration	=	'const' IDENT '=' NUM ';'
VarDeclaration	=	'var' IdentList ':' VarType ';'
IdentList	=	IDENT { ',' IDENT }
InnerBlock	=	kwBEGIN {Stmt ';'} kwEND | Stmt
Stmt	=	AssignStmt | CallStmt | IFStmt
		| WhileStmt | ForStmt | ExitStmt
		| BreakStatemement | ContinueStmt | ArrayAsignStmt
AssignStmt	=	IDENT '=' Expr
ArrayAsignStmt	=	ArrayAccess '=' Expr
CallStmt	=	IDENT '(' CallParameterList ')'
CallParameterList	=	CallParameter {COMMA CallParameter} | ε
IFStmt	=	'if' Expr 'then' InnerBlock ElseStmt
ElseStmt	=	'else' InnerBlock | ε
WhileStmt	=	'while' Expr 'do' InnerBlock
ForStmt	=	'for' IDENT '=' Expr 'to' Expr 'do' InnerBlock
		| 'for' IDENT '=' Expr 'downto' Expr 'do' InnerBlock
Expr	=	ExprLog {('AND' | 'OR') ExprLog}
ExprLog	=	ExprCmp {('=' | '!=' | '<' | '<=' | '>' | '>=') ExprCmp}
ExprCmp	=	ExprTerm {('*' | '/' | 'mod') ExprTerm}
ExprTerm	=	IDENT | CallStmt | ArrayAccess | NUM
		| 'true' | 'false' | '(' Expr ')'

2.3 Přední část překladače

V předchozím textu byla zmíněna skutečnost, že přední část překladače má za úkol přečíst zdrojový kód, zpracovat jej a ve vhodné formě výsledek předat dalším částem překladače.

Konstruovaná přední část bude zpracovávat jazyk Mila a výstupem bude abstraktní syntaktický strom ve formátu GCC GENERIC nebo LLVM IR. Tato transformace se provede v několika krocích. Nejprve bude přečten lexikálním analyzátorem znak po znaku vstupní zdrojový kód a výsledkem bude proud tokenů. Lexikální analyzátor bude implementován konečným automatem. Vzniklé tokeny se zpracují syntaktickým analyzátorem, který má za úkol instancovat abstraktní syntaktický strom. Syntaktický analyzátor bude implementovaný zásobníkovým automatem. V poslední fázi bude průchodem abstraktního syntaktického stromu vytvořena vnitřní forma překladače.

2.4 Lexikální analyzátor

Vstupní částí překladače je lexikální analyzátor. Ze vstupní posloupnosti znaků tvoří výstupní posloupnost symbolů. V této fázi se odstraňují bílé znaky a komentáře. Lexikální analyzátor jazyka Mila je realizovatelný deterministickým konečným automatem, což bude dokázáno jeho konstrukcí. Pro konstrukci lexikálního analyzátoru jsou v zásadě dvě možnosti. Lze využít nástroje typu flex [1], který kód pro lexikální analyzátor na základě popisu vygeneruje nebo lze napsat lexikální analyzátor ručně. V této práci bude lexikální analyzátor psaný ručně - aby se ukázala názornost řešení.

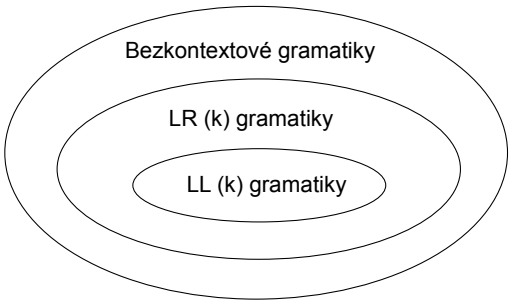
2.5 Syntaktický analyzátor

Syntaktický analyzátor bude z posloupnosti tokenů stavět abstraktní syntaktický strom. Podobně jako v případě lexikálního analyzátoru je možnost syntaktický analyzátor zkonstruovat ručně, nebo vygenerovat. Kód pro syntaktický analyzátor umí vygenerovat například nástroje yacc [12] nebo bison [8]. V tomto případě bude využita ruční konstrukce z důvodu názornosti.

Nejprve je však důležité zjistit, do které množiny jazyků patří popisovaný jazyk Mila. Z hlediska teorie formálních jazyků je zřejmé, že uvedený jazyk nebude regulárním jazykem, neboť obsahuje konstrukci begin - end, která je uvedená vždy v párech a může se neomezeně zanořovat - program může například teoreticky obsahovat neomezené množství vnořených for cyklů. Dle pumping lemma tak nemůže být regulárním jazykem, takže pro zpracování jazyka Mila nebude stačit konečný automat [10].

Jazyk Mila je bezkontextovým jazykem. Bezkontextové jazyky generují bezkontextové gramatiky, které lze rozdělit na tři množiny - viz obrázek 2.1. Podle toho, do které množiny jazyk patří, existují dvě základní metody pro syntaktickou analýzu - syntaktická analýza shora dolů pro LL(k) jazyky, nebo analýza zdola nahoru pro LR(k) jazyky. Výpočetní model je u obou shodný - zásobníkový automat [11].

V tomto případě bude konstrukcí LL(k) gramatiky dokázáno, že jazyk Mila patří do množiny LL(k) jazyků, a proto bude syntaktický analyzátor pracovat metodou shora dolů. Pro LL(k) gramatiku bude sestrojena tabulka first a follow, rozkladová tabulka a nakonec bude implementován samotný parser metodou rekurzivních procedur.



Obrázek 2.1: Bezkontextové gramatiky

[7]

2.6 Abstraktní syntaktický strom

Pro jazyk Mila bude definován specifický abstraktní syntaktický strom, který odráží jednotlivé konstrukce jazyka. Z postaveného AST potom bude průchodem vygenerován obecnější GCC GENERIC / LLVM IR. Teoreticky by šlo AST vynechat a generovat vnitřní formu překladače rovnou, ale pak by byl výsledek méně přehledný. Výhodou AST je také skutečnost, že nad ním lze před konečným překladem provádět i jiné operace - například optimalizace nebo vizualizace.

2.7 Transformace AST na vnitřní formu překladače

Po pročtení předchozího textu je zřejmé, že až do této fáze nebylo zapotřebí řešit specifičnosti GCC nebo LLVM. Od lexikální analýzy po sestavení AST bylo vše nezávislé na cíli. Aby byla práce co nejelegantnější, tak bude do AST přidána podpora pro vzor návštěvník a překlad se bude řešit pomocí dvou nezávislých návštěvnických tříd. Toto řešení má nespornou výhodu ve sdílení většiny kódu. Tato skutečnost se projeví např. při opravách chyb nebo při přidávání nových konstrukcí do jazyka. Vzor návštěvník by v budoucnu mohl mít i další uplatnění - např. pro optimalizační návštěvnické třídy.

Kapitola 3

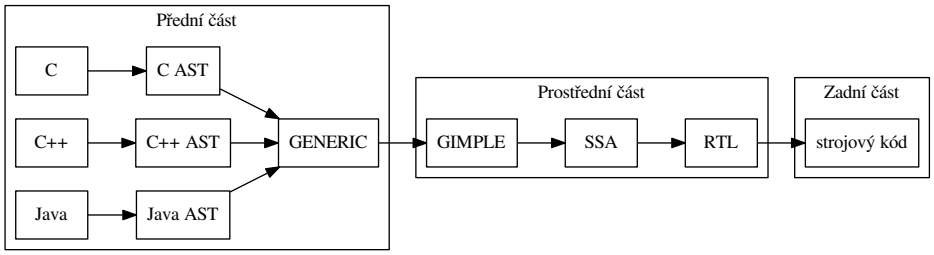
Popis rozhraní GCC a LLVM

3.1 Úvod

3.1.1 Základní informace

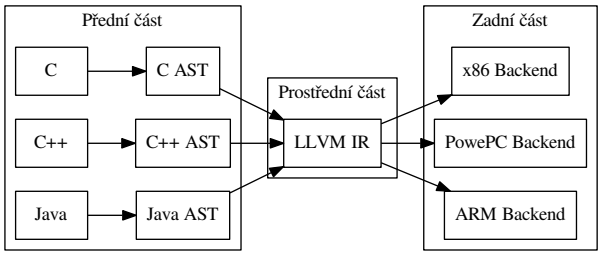
GCC i LLVM jsou rozsáhlé projekty poskytující velké množství komponent, se kterými lze pracovat [9] [5]. V této kapitole budou popsána rozhraní, pomocí kterých se generuje vnitřní forma daného překladače. A z té se následně vytvoří spustitelný kód.

Nejdříve bude obecně popsán způsob, jakým GCC a LLVM uvnitř pracují a následně budou postupně rozebrány možnosti generovaní jednotlivých konstrukcí. Popis rozhraní je omezen pouze na části, které jsou potřebné k sestrojení přední části překladače jazyka Mila.



Obrázek 3.1: Schéma GCC

[3]



Obrázek 3.2: Schéma LLVM

[14]

3.1.2 GCC

Návrh překladačů rodiny GCC ilustruje nejlépe obrázek 3.1. Ve schématu jsou vidět jednotlivé komponenty, ze kterých se překladač skládá. Nejvíce je pro tuto práci zajímavá přední část. V této části si lze povšimnout, že se AST konkrétního vstupního jazyka transformuje do formátu GENERIC. GENERIC je grafově orientovaný jazyk a lze pomocí něj reprezentovat celý program. Vytváření jednotlivých uzlů je relativně přímočaré. Kromě správného sestavení grafu je třeba ještě nastavit jednotlivým uzlům potřebné atributy. Namísto GENERIC by šlo generovat např. SSA, ale pak by práce byla složitější a pracnější, což není cílem. Generování této vnitřní formy bude v následujících kapitolách popisováno.

3.1.3 LLVM

LLVM má jednodušší třífázový návrh. Jak je vidět na obrázku 3.2, tak se od přední části předpokládá vytvoření LLVM IR. Tato forma se pak v druhé fázi několika průchody v prostřední části optimalizuje za vzniku modifikovaného LLVM IR a ve třetí fázi se z ní zadní částí generuje strojový kód.

LLVM IR je značně podobný jazyku symbolických adres [4]. V tomto jazyce jsou základním stavebním kamenem bloky. Do těchto bloků se pak generují jednotlivé instrukce. Vytváření instrukcí je podobně jako v GENERIC přímočaré. Oproti GCC GENERIC je však důležité zdůraznit, že je to jazyk nižší úrovně a pro některé konstrukce, jako např. podmíněné příkazy, se vyžaduje od přední části větší předzpracování. LLVM IR lze generovat textově nebo pomocí rozhraní LLVM. V dalším textu bude rozebíráno generování vnitřní formy pomocí rozhraní LLVM, protože je to standardní způsob jakým i ostatní přední části pracují.

3.2 Obecný popis využitých rozhraní

3.2.1 Základní informace

V této části budou obecně popsány jednotlivé dále využité funkce a datové struktury. Je to tedy úvod do samotných rozhraní.

3.2.2 GCC

Celá filozofie GENERIC tkví v reprezentaci programu pomocí grafu [9]. V tomto grafu existují vzájemně propojené uzly, které jsou typu tree. Tyto uzly jsou jazykově nezávislé a poskytují podporu pro relativně vysokoúrovňový popis programu.

Uzly je doporučeno stavět pomocí funkcí, které začínají prefixem build. Deklarace těchto funkcí je umístěna v hlavičkovém souboru ∼/gcc/tree.h a samotná implementace se nachází v souboru ∼/gcc/tree.c. Tyto zdrojové soubory jsou vhodným studijním materiálem pro tvorbu přední části, protože je zde kód dobře dokumentován.

3.2.3 LLVM

LLVM IR je mezijazyk, podobný JSA. Tento mezijazyk obsahuje funkce, bloky a instrukce [4]. Pro uložení hodnot slouží nekonečný počet registrů a pamět, ke které se přistupuje pomocí Load a Store instrukcí.

Kontejnerem pro funkce je z hlediska přední části třída Module. Funkce je typu Function, blok typu BasicBlock a instrukce jsou podtypem třídy Instruction. Každá funkce může mít v těle libovolný počet bloků, do kterých jsou instrukce generovány třídou IRBuilder. Pro studium těchto rozhraní nad rámec této práce je doporučena dokumentace na oficiálních stránkách projektu LLVM.

3.3 Inicializace prostředí

3.3.1 Základní informace

Aby bylo možné využívat dále popisované rozhraní, tak je třeba mít stažené zdrojové kódy využitého překladače a dále do projektu začlenit vyvíjenou přední část. Pro prvotní nastavení lze vyjít ze šablon v příloze, které obsahují minimální přední části generující program vracející po spuštění hodnotu nula. Tyto přední části jsou minimální zkompilovatelnou přední částí pro GCC a LLVM.

3.3.2 GCC

Přední část využívající GCC je těsně svázána se zbytkem projektu - důvodem je monolitický návrh překladače. Všechny potřebné struktury jsou již inicializovány a přední část tak může začít zpracovávat kód. Adresář obsahující vzorovou přední část se doporučuje začlenit do projektu do adresáře ∼/gcc/sfe.

3.3.3 LLVM

LLVM se používá modulárně, tudíž není přední část tak moc svázaná se zbytkem projektu. Komponenty LLVM se používají jako knihovny a mají mezi sebou minimální vazby. U LLVM přední části vycházející ze šablony je vidět, že se musí nejprve vytvořit instance IRBuilder a Module.

```
1 IRBuilder<> builder(getGlobalContext());
2 Module * module = new Module("Sfe", getGlobalContext());
```

Zdrojový kód 3.1: Inicializace LLVM

IRBuilder je pomocná třída, která umožňuje generovat LLVM IR kód a Module je kontejner pro veškeré funkce, globální proměnné a konstanty v programu. Minimální přední část z přílohy se doporučuje začlenit do složky ∼/projects/sfe.

3.3.4 Shrnutí

Hlavní rozdíl v překladačích GCC a LLVM je monolitická versus modulární architektura. Tento fakt je dán mimo jiné také dobou, kdy oba projekty vznikaly. Pro vývoj přední části stačí ale v obou případech vhodně začlenit šablonu a lze začít generovat kód.

3.4 Deklarace funkce

3.4.1 Základní informace

Funkce je v obou formách kontejnerem pro spustitelný kód, proto se zde popis její tvorby vyskytuje na prvním místě. Nechť je pro generování uvažována následující funkce zapsaná v jazyku C.

```
1 int main()
2 {
3   return 0;
4 }
```

Zdrojový kód 3.2: Generovaná funkce

3.4.2 GCC

Zdrojový kód 3.3 pro vytvoření funkce je oproti dalším konstrukcím relativně dlouhý, ale v případě GCC není složitý na pochopení.

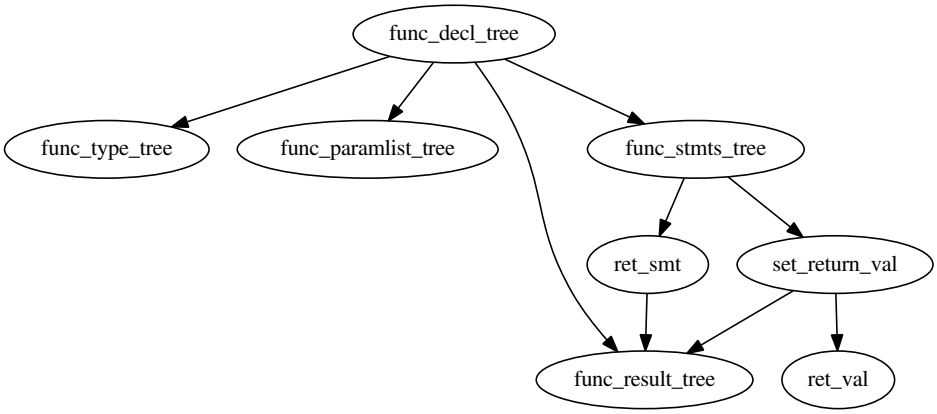
Nejprve je třeba na řádku 1 vytvořit uzel obsahující typ návratové hodnoty. Návratová hodnota bude typu celé číslo. Dále lze vytvořit uzel reprezentující danou funkci. Při vytváření uzlu funkce se jako argument předá uzel reprezentující typ návratové hodnoty a název generované funkce. Vytvořenému uzlu se musí nastavit potřebné parametry - funkce je statická, veřejná a globální. Pak bude chápána jako funkce v úvodní ukázce.

```
1  tree func_type_tree = build_function_type_list(integer_type_node,
       NULL_TREE);
2  tree func_decl_tree = build_decl(BUILTINS_LOCATION, FUNCTION_DECL,
       get_identifier("main"), func_type_tree);
3  TREE_STATIC(func_decl_tree) = true;
4  TREE_PUBLIC(func_decl_tree) = true;
5  DECL_CONTEXT(func_decl_tree) = NULL_TREE;
6
7  tree func_paramlist_tree = NULL_TREE;
8  DECL_ARGUMENTS(func_decl_tree) = (func_paramlist_tree = nreverse(
       func_paramlist_tree));
9
10 tree func_result_tree = build_decl(BUILTINS_LOCATION, RESULT_DECL,
       NULL_TREE, TREE_TYPE(func_type_tree));
11 DECL_CONTEXT(func_result_tree) = func_decl_tree;
12 DECL_ARTIFICIAL(func_result_tree) = true;
13 DECL_IGNORED_P(func_result_tree) = true;
14 DECL_RESULT(func_decl_tree) = func_result_tree;
15
16 tree func_art_block_tree = build_block(NULL_TREE, NULL_TREE, NULL_TREE,
       NULL_TREE);
17 DECL_INITIAL(func_decl_tree) = func_art_block_tree;
18
19 tree func_stmts_tree = alloc_stmt_list();
20
21 tree ret_val = build_int_cstu(integer_type_node, 0);
22 tree set_return_val = build2(MODIFY_EXPR, TREE_TYPE(func_result_tree),
       func_result_tree, ret_val);
23 append_to_statement_list(set_return_val, &func_stmts_tree);
24
25 tree ret_smt = build1(RETURN_EXPR, void_type_node, func_result_tree);
26 append_to_statement_list(ret_smt, &func_stmts_tree);
27
28 DCL_SAVED_TREE(func_decl_tree) = build3(BIND_EXPR, void_type_node,
       NULL_TREE, func_stmts_tree, func_art_block_tree);
29
30 gimplify_function_tree(func_decl_tree);
31 cgraph_node::finalize_function(func_decl_tree, false);
```

Zdrojový kód 3.3: Funkce v GENERIC

Na řádce 7 je nutno vytvořit uzel reprezentující parametry funkce a přiřadit ho dané funkci. Generování parametrů bude popsáno v jedné z dalších podkapitol. Zde bude uvažována funkce bez parametrů.

V GENERIC se dále musí na řádku 10 vytvořit uzel, který reprezentuje návratovou hodnotu. Pokud bude později potřeba, aby se funkce ukončila a vrá-



Obrázek 3.3: Příklad funkce - GENERIC

tila hodnotu, tak bude zapotřebí nejprve do tohoto vytvořeného uzlu vracenou hodnotu přiřadit. Po přiřazení lze provést příkaz reprezentující vyskočení z funkce.

Na řádku 16 je vytvářen uzel pro vstupní blok funkce. Do vstupního bloku funkce se dále přiřadí uzel se seznamem příkazů. Tento uzel je kontejnerem pro veškeré příkazy v těle funkce. Pro ukázkovou funkci se do něj vloží příkaz, který změní návratovou hodnotu na nulu. Po nastavení návratové hodnoty lze do těla příkazů vložit uzel reprezentující příkaz return, který hodnotu vrátí volající funkci.

Tímto je vytvořen graf funkce. Tento graf je třeba po vytvoření poslat k další transformaci prostřední části překladače. Z deklarace funkce je vidět, že se k reprezentaci všech uzlů používá struktura tree. Pro názornou ilustraci je na obrázku 3.3 vidět vizuální reprezentace vytvořeného grafu funkce.

3.4.3 LLVM

Pro LLVM přední část začíná ve zdrojovém kódu 3.4 vytvoření funkce tím, že se na řádku 1 vytvoří instance podtřídy Type obsahující návratový typ.

```
1 Type * returnType = Type::getInt32Ty(getGlobalContext());
2 std::vector<Type*> argTypes;
3 FunctionType * functionType = FunctionType::get(returnType, argTypes,
      false);
4 Function * function = Function::Create(functionType, Function::
      ExternalLinkage, "main", module);
5 BasicBlock *entryBB = BasicBlock::Create(getGlobalContext(), "entry",
      function);
6 builder.SetInsertPoint(entryBB);
7 Value * returnValue = builder.getInt32(0);
8 Value * returnInst = builder.CreateRet(returnValue);
```

Zdrojový kód 3.4: Funkce v LLVM

Dále je třeba deklarovat vektor typů parametrů. Tento vektor nebude obsahovat žádné prvky, protože je tvořena funkce bez parametrů. Na řádku 4 je vytvářena samotná hlavička funkce.

Pokud je chtěná i deklarace těla funkce, tak se musí na řádku 5 vytvořit instance třídy BasicBlock, která reprezentuje blok příkazů.

Zde je důležité zdůraznit skutečnost, že basic blok musí mít právě jeden vstupní a jeden výstupní bod. Jak se ukáže dále, tak pro účely ovlivňování toku programu v těle funkce např. podmínkami nebo cykly, bude nutné do těla funkce generovat několik basic bloků, které na sebe budou odkazovat.

Pro generování jednotlivých instrukcí se využije instance třídy IRBuilder, které se musí před použitím nastavit místo pro vkládání instrukcí. V tomto případě se nastaví místo vkládání na vytvořený basic blok.

Nyní lze začít vkládat jednotlivé instrukce. V tomto příkladu se na řádku 8 vloží pouze Ret instrukce vracející konstantu nula.

Pro LLVM verzi je reprezentace v paměti lehce odlišná, jak dokládá obrázek 3.4. LLVM má výhodu, že lze zapsat uvedený kód textově přímo v LLVM IR. Pro ilustraci je zde tento ekvivaletní textový zápis ukázán jako zdrojový kód 3.5. Tento textový zápis si umí LLVM načíst a vyrobit z něj instanci třídy Module.

```
1 ; ModuleID = 'Sfe'
2
3 define i32 @main() {
4 entry:
5   ret i32 0
6 }
```

Zdrojový kód 3.5: LLVM IR

3.4.4 Shrnutí

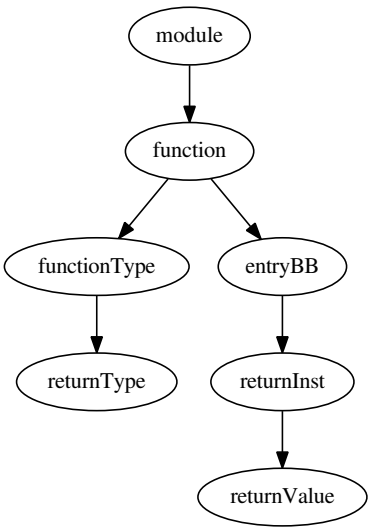
Byla vytvořena jednoduchá funkce a zároveň bylo prakticky ukázáno, jak lze pracovat s rozhraním obou překladačů. Generovaná funkce je ve skutečnosti minimální zkompilovatelný program. Popisovaná funkce je součástí minimálních předních částí v příloze.

3.5 Proměnné

3.5.1 Základní informace

Deklarace proměnné je základní věc z pohledu imperativních programovacích jazyků. Proto je důležité vědět, jak ji lze vygenerovat.

Proměnné se budou deklarovat v rámci funkce, budou tedy lokální, a bude se od nich očekávat standardní chování - bude chtěné aby šlo hodnotu proměnné kdykoli číst nebo měnit.



Obrázek 3.4: Příklad funkce - LLVM IR

3.5.2 GCC

Zdrojový kód 3.6 ukazuje, jak lze deklarovat proměnnou typu celé číslo a následně s ní pracovat. Nejprve je třeba vytvořit na řádku 3 deklaraci proměnné x.

```
1  tree func_decl_tree = ...;
2
3  tree var_decl_tree = build_decl(BUILTINS_LOCATION, VAR_DECL,
       get_identifier("x"), integer_type_node);
4  DECL_CONTEXT(var_decl_tree) = func_decl_tree;
5  TREE_USED(var_decl_tree) = true;
6  DECL_INITIAL(var_decl_tree) = build_int_cstu(integer_type_node, 0);
7  TREE_STATIC(var_decl_tree) = true;
8
9  tree int_ptr_type = build_pointer_type(integer_type_node);
10 tree var_ptr_tree = build1(ADDR_EXPR, int_ptr_type, var_decl_tree);
11
12 tree var_deref_tree = build1(INDIRECT_REF, integer_type_node,
       var_ptr_tree);
```

Zdrojový kód 3.6: GENERIC proměnné

U deklarované proměnné se nastaví kontext na danou funkci a inicializuje se hodnotou nula. Uzel reprezentující proměnnou lze pak hned používat. Vzniklou proměnnou lze v těle funkce kdykoliv číst nebo měnit.

Zajímavé je ještě ukázat, jak lze na proměnnou získat ukazatel. Tato funkčnost je v GENERIC implementována typem uzlu ADDR_EXPR. Uzel var_ptr_tree na řádku 10 obsahuje ukazatel na danou proměnnou. Opačnou operací je dereference ukazatele a získání hodnoty proměnné. Dereference je v GENERIC implementována typem uzlu INDIRECT_REF. Uzel var_deref_tree na řádku 12 tak obsahuje hodnotu původní proměnné.

3.5.3 LLVM

LLVM pracuje s registry a vyžaduje u nich SSA formu a to znamená, že hodnoty registrů nejsou měnitelné. Pokud by byl implementován například funkcionální jazyk, který nemá měnitelné proměnné, tak by to nebyl problém. V imperativním jazyku to však problém je. Pro ilustraci je zde uveden zdrojový kód 3.7.

```
1 if (a~> 0)
2   x = 1;
3 else
4   x = 0;
5 return x;
```

Zdrojový kód 3.7: Podmíněné přiřazení

Tento kód by se nedal pomocí SSA registrů realizovat. Vzniklo by totiž něco podobného jako zdrojový kód 3.8. U příkazu return není známo kterou hodnotu vrátit.

```
1 if (a~> 0)
2   x.1 = 1;
3 else
4   x.2 = 0;
5 return x.1 ? x.2;
```

Zdrojový kód 3.8: Podmíněné přiřazení SSA

LLVM poskytuje dvě možnosti, jak problém vyřešit. První variantou je používat tzv. Φ instrukce, které vrátí správnou hodnotu na základě výsledku předchozí podmínky. Tato možnost ale dále v práci nebude rozvíjena, protože pokud by byla konstruována přední část, která používá Φ instrukce, tak by se tvorba přední části překladače značně zkomplikovala [6]. V práci bude rozebírána druhá varianta - místo využití SSA registru se bude pomocí Alloca instrukce pro každou proměnnou alokovat paměť. LLVM pak umožňuje pomocí Load a Store instrukcí libovolný počet čtení / zápisu.

Tento přístup je oproti GCC komplikovanější, protože většina ostatních instrukcí vyžaduje jako argument SSA registr. Před použitím se tak musí nejprve do SSA registru načíst aktuální hodnota pomocí Load instrukce a následně lze tento registr předat dále. Podobně je třeba pomocí Store instrukce hodnotu ukládat při změně.

Pro ilustraci uvedeného práce s proměnnou poslouží zdrojový kód 3.9.

```
1 Value * decl = builder.CreateAlloca(Type::getInt32Ty(getGlobalContext())
      , 0, "x");
2
3 Value * val = builder.CreateLoad(decl, "x");
4
5 Value * newVal = ...
6 builder.CreateStore(decl, newVal);
```

Zdrojový kód 3.9: LLVM proměnné

V LLVM se tedy nejprve na řádku 1 pro proměnnou x alokuje paměť. S výsledným ukazatelem se pak dále pracuje. Před použitím se musí na řádku 4 hodnota načíst. Pak lze nově vytvořený SSA registr použít např. jako argument instrukce Add, která sečte hodnoty dvou registrů. Pro změnu je třeba hodnotu ukládat - jak ilustruje řádek 6.

V přechozí části, která pojednávala o proměnných v GCC, byly rozebrány mimo jiné dvě operace: vytvoření ukazatele na proměnnou a dereference ukazatele. V LLVM lze získat ukazatel jen na předem alokované paměťové místo. Dereference se pak provádí Load instrukcí. Na registr ukazatel získat nelze.

3.5.4 Shrnutí

GCC umožňuje pracovat s proměnnými jednoduše a u LLVM je třeba mít na paměti omezení IR. Při práci s proměnnými se projevuje nízkoúrovňovost LLVM IR.

3.6 Parametry funkcí

3.6.1 Základní informace

Byla rozebrána deklarace funkce a práce s proměnnými. Ještě však nebylo zmíněno jak do deklarace funkce zavést jednotlivé parametry. Bude ukázáno jak deklarovat parametry funkce tak, aby se s nimi v těle funkce dalo pracovat jako se standardními proměnnými. Tato část bude vycházet z podkapitoly zabývající se deklaraci funkcí a bude tak rozšířena. Pro následující popis je uvažován příklad funkce, která má parametr x typu celé číslo.

3.6.2 GCC

V přední části pro GCC stačí v místě, kde se deklaruje hlavička funkce, dodat daný parametr.

```
1  tree func_decl_tree = ...;
2
3  tree func_paramlist_tree = NULL_TREE;
4
5  tree arg_type_tree = signed_type_for(integer_type_node);
6  tree arg_tree = build_decl(BUILTINS_LOCATION, PARM_DECL, "x",
       arg_type_tree);
7  DECL_ARG_TYPE(arg_tree) = TREE_TYPE(arg_tree);
8  DECL_CONTEXT(arg_tree) = func_decl_tree;
9
10 func_paramlist_tree = chainon(arg_tree, func_paramlist_tree);
11
12 DECL_ARGUMENTS(func_decl_tree) = (func_paramlist_tree = nreverse(
       func_paramlist_tree));
```

Zdrojový kód 3.10: GCC parametry

Proměnná arg_tree pak obsahuje daný parametr, se kterým lze pracovat jako se standardní proměnnou.

3.6.3 LLVM

U LLVM je situace komplikovanější. Nejprve se podobně jako v GCC deklaruje vektor typů parametrů a vytvoří se na řádku 6 hlavička funkce - jak ukazuje zdrojový kód 3.11.

```
1  std::vector<Type*> argTypes;
2
3  Type * argType = Type::getInt32Ty(getGlobalContext());
4  argTypes.push_back(argType);
5
6  FunctionType * functionType = FunctionType::get(returnType, argTypes,
       false);
7
8  Type * varType = Type::getInt32Ty(getGlobalContext());
9  AllocaInst * var = builder.CreateAlloca(varType, 0, "x");
10
11 Function::arg_iterator AI = function->arg_begin();
12 AI->setName("x");
13 builder.CreateStore(AI, val);
```

Zdrojový kód 3.11: LLVM parametry

Po vytvoření hlavičky funkce se parametry projdou, pojmenují se a alokuje se pro každý parametr paměť, do které se zkopíruje hodnota parametru.

Alokování paměti pro novou proměnnou se dělá kvůli možnosti měnit hodnotu parametru uvnitř těla funkce. Proměnnou var lze pak používat jako každou jinou proměnnou v rámci LLVM.

3.6.4 Shrnutí

Deklarace parametrů funkce je v GCC přímočará, v LLVM se musí navíc alokovat nová proměnná, aby šlo s parametrem pracovat jako s lokální proměnnou.

3.7 Aritmeticko-logické výrazy

3.7.1 Základní informace

V této podkapitole budou rozebrány aritmetické a logické výrazy používající binární operátory. Výsledkem výrazu bude vždy hodnota typu celé číslo a stejného typu budou i oba operandy.

Tabulka 3.1: Typy binárních operátorů - LLVM IR a GENERIC

Operátor	Typ uzlu GENERIC	Metoda IRBuilder LLVM
+	PLUS_EXPR	CreateAdd
-	MINUS_EXPR	CreateSub
	MULT_EXPR	CreateMul
/	TRUNC_DIV_EXPR	CreateSDiv
AND	TRUTH_AND_EXPR	CreateAnd
OR	TRUTH_OR_EXPR	CreateOr
==	TRUTH_EQ_EXPR	CreateICmpEQ
!=	TRUTH_NEQ_EXPR	CreateICmpNE
<	LT_EXPR	CreateICmpSLT
<=	LE_EXPR	CreateICmpSLE
>	GT_EXPR	CreateICmpSGT
>=	GTE_EXPR	CreateICmpSGE

3.7.2 GCC

Vytvoření uzlu rezprezentující sečtení dvou čísel ukazuje zdrojový kód 3.12.

```
1 tree ltree = ...;
2 tree rtree = ...;
3 tree result = build2(PLUS_EXPR, TREE_TYPE(ltree), ltree, rtree);
```

Zdrojový kód 3.12: GCC výrazy

GENERIC řeší binární operace funkcí build2, které se předá typ operátoru, typ výsledku a levý a pravý operand. Výsledkem je nový uzel reprezentující výslednou hodnotu. Pro úplnost jsou uvedeny standardní typy binárních operátorů a odpovídající typ uzlu v GENERIC v tabulce 3.1. Ostatní podporované operátory lze najít v dokumentaci GCC.

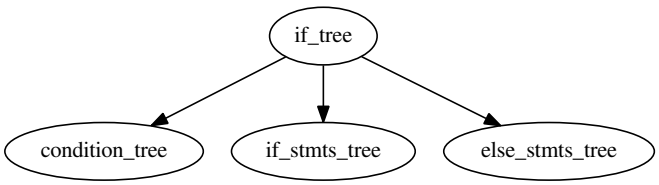
3.7.3 LLVM

LLVM používá IRBuilder ke vkládání jednotlivých instrukcí provádějící danou operaci. Oba operandy jsou SSA registry a výsledkem je nově vzniklý SSA registr obsahující výslednou hodnotu. Například sečtení dvou čísel ilustruje zdrojový kód 3.13.

```
1 Value * leftVal = ...;
2 Value * rightVal = ...;
3 Value * result = builder.CreateAdd(leftVal, rightVal, "addtmp");
```

Zdrojový kód 3.13: LLVM výrazy

Proměnné leftVal a rightVal reprezentují SSA registry, které obsahují hodnoty typu celé číslo a poslední argument metody CreateAdd obsahuje název nově vzniklého registru. Tento název sice není nutný pro LLVM ale je užitečný pro člověka, který zkoumá vygenerovaný IR kód. Vhodný název se hodí zejména pro účely ladění přední části.



Obrázek 3.5: Podmíněný příkaz - GENERIC

Pro úplnost jsou metody pro vytváření základních aritmeticko-logických instrukcí uvedeny v tabulce 3.1. Další metody pro vkládání aritmeticko-logických instrukcí jsou popsány v dokumentaci třídy IRBuilder.

3.7.4 Shrnutí

Aritmeticko-logické příkazy lze v GCC a LLVM implementovat podobně jednoduše. Podporována je jich celá řada. Lze je do sebe skládat a použít je bezproblémově.

3.8 Podmíněné příkazy

3.8.1 Základní informace

Pro tuto podkapitolu bude cílem vygenerovat jednoduchý podmíněný příkaz s alternativní částí.

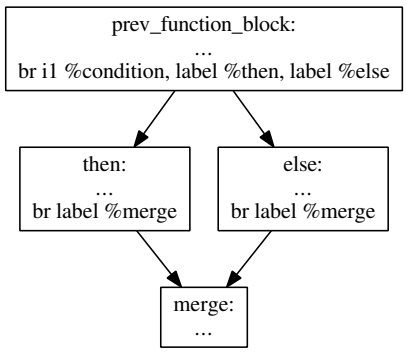
3.8.2 GCC

GENERIC umožňuje velmi jednoduchou implementaci podmíněného příkazu, jak dokazuje zdrojový kód 3.14.

```
1 tree condition_tree = ...;
2 tree if_stmts_tree = ...;
3 tree else_stmts_tree = ...;
4 tree if_tree = build3(COND_EXPR, void_type_node, condition_tree,
      if_stmts_tree, else_stmts_tree);
```

Zdrojový kód 3.14: GCC podmíněný příkaz

Uzel pro podmíněný příkaz se vytvoří voláním funkce build3, který v závislosti na podmínce uložené v condition_tree provede příkazy uložené v if_stmts_tree, nebo else_stmts_tree. Pokud je else_stmts_tree NULL, pak se alternativní větev nebude vůbec generovat. Pro ilustraci je zde na obrázku 3.5 uveden výsledný strom.



Obrázek 3.6: Podmíněný příkaz - LLVM IR

3.8.3 LLVM

V implementaci podmíněného příkazu pro LLVM nastávají komplikace s nutností generovat správně basic bloky.

Pokud se má v půlce funkce vygenerovat podmíněný příkaz, tak se musí aktuálně vytvářený basic blok ukončit. Ukončí se instrukcí, reprezentující podmíněný skok. V závislosti na splnění či nesplnění podmínky bude skočeno do jednoho ze dvou nově vytvořených basic bloků. Tyto nově vytvořené basic bloky budou reprezentovat část pro splněnou podmínku a alternativní část. Nakonec se musí vygenerovat nový slučovací basic blok, do kterého se skočí poslední instrukcí z basic bloků reprezentujících větve podmíněného příkazu.

```
1  BasicBlock *thenBB = BasicBlock::Create(getGlobalContext(), "then",
       function);
2  BasicBlock *elseBB = BasicBlock::Create(getGlobalContext(), "else");
3  BasicBlock *mergeBB = BasicBlock::Create(getGlobalContext(), "merge");
4
5  Value * condition = ...;
6  builder.CreateCondBr(condition, thenBB, elseBB);
7
8  builder.SetInsertPoint(thenBB);
9  ...
10 builder.CreateBr(mergeBB);
11
12 function->getBasicBlockList().push_back(elseBB);
13 builder.SetInsertPoint(elseBB);
14 ...
15 builder.CreateBr(mergeBB);
16
17 elseBB = builder.GetInsertBlock();
18 function->getBasicBlockList().push_back(mergeBB);
19 builder.SetInsertPoint(mergeBB);
```

Zdrojový kód 3.15: LLVM podmíněný příkaz

Zdrojový kód 3.15 ukazuje popsanou funkčnost. Nejprve se vytvoří všechny zmíněné basic bloky. Následně se vygeneruje do existujícího basic bloku podmíněný skok, který je reprezentován instrukcí Br. V závislosti na podmínce

potom tato instrukce provede skok do then nebo else části. Touto instrukcí je na řádku 6 aktuální basic blok korektně ukončen.

Nyní je třeba nastavit builder, aby generoval instrukce do then části. Poté se vygeneruje podmíněný kód na řádku 9 a basic blok se ukončí skokem do merge části.

Vloží se else basic blok a podobně jako then část se vygeneruje. Zase je třeba tento blok ukončit skokem do merge části.

Nakonec se vloží na řádku 18 merge basic blok a nastaví se builder aby generoval další instrukce do něj. Tímto postupem byl vygenerován podmíněný příkaz a zároveň nebyl pokažen tok programu. Výsledný IR kód ilustruje obrázek 3.6.

3.8.4 Shrnutí

GENERIC je na rozdíl od LLVM IR v otázkách řízení toku programu mnohem přímočařejší. V GCC stačí pro implementaci podmíněného příkazu vytvořit správný uzel, v LLVM IR se musí generovat jednotlivé basic bloky a korektně je ukončovat.

3.9 Cykly

3.9.1 Základní informace

V následujícím textu bude konstruován klasický for cyklus, který obsahuje inicializační část, část kde se testuje podmínka a část pro provedení kódu po iteraci. U toho cyklu bude následně ukázáno, jak implementovat break a continue příkazy. Pro další popis je uvažován for cyklus uvedený ve zdrojovém kódu 3.16.

```
1 for (i = 0; i < j; i++)
2 {
3   ...
4 }
```

Zdrojový kód 3.16: for cyklus

3.9.2 GCC

Stavbu uvedeného for cyklu ilustruje zdrojový kód 3.17. Nejprve se na řádcích 1 až 7 vytvoří uzly, do kterých budou vkládány příkazy, a inicializuje se řídící proměnná. Toto se děje před vstupem do cyklu.

V těle cyklu se na řádcích 9 až 12 otestuje podmínka pro provedení iterace. Aby byl dodržen výše uvedený příklad, tak se zde vyrobí podmínka i $<$ j. Pokud podmínka nebude splněna, tak se vyskočí z těla cyklu uzlem typu EXIT_EXPR. Tímto typem uzlu je reprezentován příkaz i příkaz break.

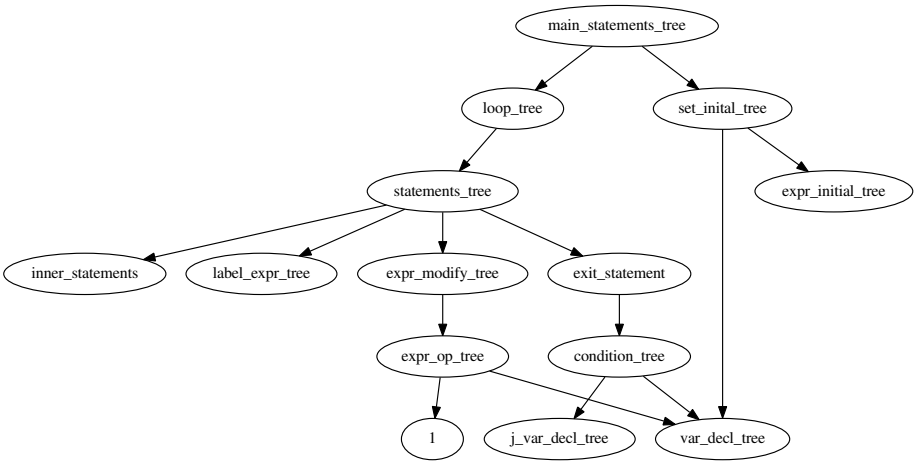
Pokud je chtěná podpora příkazu continue, tak se musí na řádcích 14 až 17 vyrobit návěstí. Návěstí je v GENERIC uzel, který označuje místo v kódu, na který lze z jiného místa skočit. Je to tedy podobné jako v jiných jazycích. Zde se však pouze vygeneruje uzel, který se nikam ještě nepřidává.

Na řádku 19 se vygenerují příkazy, které se mají provést v těle cyklu. Po vygenerování příkazů uvnitř těla cyklu se vloží zmíněné návěstí pro continue do těla cyklu. Následně se do těla cyklu vloží část, která se má provést po provedení iterace. V tomto případě to bude inkrementace proměnné i.

Když je tělo cyklu postaveno, tak se vygeneruje uzel typu LOOP_EXPR, který reprezentuje popisovaný cyklus. Výsledkem je uzel main_statements_tree, jenž obsahuje celý vygenerovaný for cyklus. Nejlépe je struktura vidět na obrázku 3.7. Tímto je for cyklus vytvořen a uzel pak lze vložit do těla generované funkce.

```
1  tree main_statements_tree = alloc_stmt_list();
2  tree statements_tree = alloc_stmt_list();
3
4  tree var_decl_tree = ...;
5  tree initial_val_tree = build_int_cst(integer_type_node, 0);
6  tree set_inital_tree = build2(MODIFY_EXPR, TREE_TYPE(var_decl_tree),
       var_decl_tree, initial_val_tree);
7  append_to_statement_list(set_inital_tree, &main_statements_tree);
8
9  tree j_var_decl_tree = ...;
10 tree condition_tree = build2(GT_EXPR, TREE_TYPE(var_decl_tree),
       var_decl_tree, j_var_decl_tree);
11 tree exit_statement = build1(EXIT_EXPR, void_type_node, condition_tree);
12 append_to_statement_list(exit_statement, &statements_tree);
13
14 tree label_decl_tree = build_decl(BUILTINS_LOCATION, LABEL_DECL,
       get_identifier("label1"), TREE_TYPE(void_type_node));
15 DECL_CONTEXT(label_decl_tree) = func_decl_tree;
16 DECL_IGNORED_P(label_decl_tree) = 1;
17 tree label_expr_tree = build1(LABEL_EXPR, void_type_node,
       label_decl_tree);
18
19 tree inner_statements = ...;
20
21 append_to_statement_list(inner_statements, &statements_tree);
22 append_to_statement_list(label_expr_tree, &statements_tree);
23
24 tree expr_op_tree = build2(PLUS_EXPR, TREE_TYPE(var_decl_tree),
       var_decl_tree, build_int_cst(integer_type_node, 1));
25 tree expr_modify_tree = build2(MODIFY_EXPR, TREE_TYPE(var_tree),
       var_tree, expr_op_tree);
26 append_to_statement_list(expr_modify_tree, &statements_tree);
27
28 tree loop_tree = build1(LOOP_EXPR, void_type_node, statements_tree);
29 append_to_statement_list(loop_tree, &main_statements_tree);
```

Zdrojový kód 3.17: GCC for cyklus



Obrázek 3.7: For cyklus - GENERIC

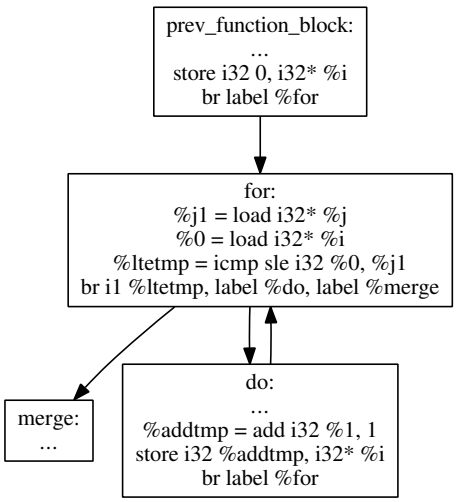
3.9.3 LLVM

Po pročtení předchozích podkapitol je patrné, že v LLVM bude for cyklus implementován několika basic bloky, mezi kterými se bude skákat. Ze zdrojového kódu 3.18 je vidět, že se vytvoří celkem tři basic bloky.

```
1  BasicBlock *forBB = BasicBlock::Create(*context, "for", function);
2  BasicBlock *doBB = BasicBlock::Create(*context, "do");
3  BasicBlock *mergeBB = BasicBlock::Create(*context, "merge");
4
5  Value * var_declaration = ...;
6  Value * initial = ...;
7  builder.CreateStore(initial, var_declaration);
8  builder.CreateBr(forBB);
9
10 builder.SetInsertPoint(forBB);
11 Value * var_declaration_j = ...;
12 Value * condition = builder.CreateICmpSLE(builder.CreateLoad(
       var_declaration), builder.CreateLoad(var_declaration_j), "ltetmp");
13 builder.CreateCondBr(condition, doBB, mergeBB);
14
15 function->getBasicBlockList().push_back(doBB);
16 builder.SetInsertPoint(doBB);
17 ...
18 Value * newVal =  builder.CreateAdd(builder.CreateLoad(var_declaration),
        builder.getInt32(1), "addtmp");
19 builder.CreateStore(newVal, var_declaration);
20 builder.CreateBr(forBB);
21
22 function->getBasicBlockList().push_back(mergeBB);
23 builder.SetInsertPoint(mergeBB);
```

Zdrojový kód 3.18: LLVM for cyklus

Ještě před samotným generováním nových bloků cyklu se ale musí vytvořit instrukce pro prvotní nastavení řídící proměnné. Dále lze aktuální basic blok



Obrázek 3.8: For cyklus - LLVM IR

ukončit skokem do for basic bloku. Následně je nastaven builder, aby další instrukce generoval do tohoto nového basic bloku.

Ve for basic bloku se otestuje podmínka a pokud platí, tak se skočí do basic bloku do jinak je proveden skok do merge basic bloku. Následně se vygeneruje do basic blok a na řádku 17 by se vložily instrukce, které mají být v těle cyklu.

Po provedení instrukcí v těle cyklu se musí aktualizovat řídící proměnná. Přičte se k ní tedy konstanta jedna. Nakonec se ukončí do basic blok skokem na for basic blok. Poslední co zbývá je vložit merge basic blok a nastavit builder aby generoval instrukce do něj.

Cyklus je vygenerován a obrázek 3.8 znázorňuje vizualizaci jednotlivých basic bloků. Pokud by bylo třeba vytvořit continue příkaz, tak stačí vytvořit skok na for basic blok, break příkaz by pak odpovídal skoku na merge basic blok.

3.9.4 Shrnutí

Implementace cyklů v LLVM vyžaduje tvorbu basic bloků a skoků, není zde žádná explicitní instrukce reprezentující cyklus. GCC GENERIC jde o úroveň výše a poskytuje podporu pro uzel typu LOOP_EXPR.

3.10 Volání funkcí

3.10.1 Základní informace

Pro následující text je uvažovaná deklarovaná funkce test.

```
1 int test(int param1, int param2);
```

Zdrojový kód 3.19: Volaná funkce

Cílem je popsat zavolání funkce standardním způsobem.

```
1 res = test(param1, param2);
```

Zdrojový kód 3.20: Volání funkce

3.10.2 GCC

Pro zavolání funkce je třeba nejprve vyrobit vektor, který se naplní hodnotami poslanými funkci. Samotné zavolání funkce lze pak vygenerovat pomocí funkce build_call_expr_loc_array, které se předá volaná funkce a argumenty. V proměnné result_tree je pak k dispozici výsledek volání.

```
1 tree param1_tree = ...;
2 tree param2_tree = ...;
3
4 std::vector<tree> parameters_vec;
5 parameters_vec.push_back(param1_tree);
6 parameters_vec.push_back(param2_tree);
7
8 tree called_function_tree = ...;
9 tree result_tree = build_call_expr_loc_array(BUILTINS_LOCATION,
      called_function_tree, parameters_vec.size(), parameters_vec.data());
```

Zdrojový kód 3.21: GCC - zavolání funkce

3.10.3 LLVM

Obdobně jako v GCC je v případě LLVM vyroben vektor, který se naplní hodnotami poslanými funkci.

```
1 Value* param1 = ...;
2 Value* param2 = ...;
3
4 std::vector<Value*> parameters;
5 parameters.push_back(param1);
6 parameters.push_back(param2);
7
8 Function * called_function = ...;
9 Value * result = builder.CreateCall(called_function, parameters, "
      calltmp");
```

Zdrojový kód 3.22: LLVM - zavolání funkce

Zavolání se provede vložením instrukce Call a v proměnné result je pak k dispozici výsledek volání.

3.10.4 Shrnutí

Volání funkcí se realizuje v GENERIC i LLVM IR podobně. Pro zavolání stačí mít odkaz na volanou funkci, vytvořit vektor obsahující argumenty a pomocí rozhraní vytvořit volání.

3.11 Pole

3.11.1 Základní informace

Tato podkapitola bude pojednávat o deklaraci pole fixní velikosti a přístupu k jeho jednotlivým prvkům.

3.11.2 GCC

Deklarace pole je podobná jako deklarace skalární proměnné. Rozdíl je pouze v zadaném typu - místo proměnné typu celé číslo se deklaruje proměnná typu pole celých čísel zadané délky. Pro následující popis bude uvažován zdrojový kód 3.23. Nejprve se na řádku 1 vytváří uzel označující typ pole celých čísel o deseti prvcích. Dále se na řádcích 4 až 8 vytváří samotné pole.

```
1  tree arr_dimension_tree = build_index_type(size_int(10));
2  tree arr_type_tree = build_array_type(integer_type_node,
       arr_dimension_tree);
3
4  tree var_decl_tree = build_decl(BUILTINS_LOCATION, VAR_DECL, "items",
       arr_type_tree);
5  TREE_STATIC(var_decl_tree) = true;
6  TREE_PUBLIC(var_decl_tree) = true;
7  DECL_CONTEXT(var_decl_tree) = func_decl_tree;
8  TREE_USED(var_decl_tree) = true;
9
10 tree index_tree = build_int_cst(5);
11 tree array_access_tree = build2(ARRAY_REF, integer_type_node, var_tree,
       index_tree);
```

Zdrojový kód 3.23: GCC - pole

Pro přístup k prvkům pole je v GENRIC implementován uzel typu ARRAY_REF. Tento uzel se vytvoří s odkazem na dané pole a index prvku. K tomuto uzlu se pak lze chovat jako k běžné proměnné - tzn. lze číst a zapisovat hodnotu. Vytvoření tohoto uzlu je ukázáno na řádcích 10 až 11, přistupuje se zde k prvku s indexem 5.

3.11.3 LLVM

Skalární proměnné se v LLVM deklarují jako ukazatele do paměti a pole budou deklarována obdobně. Nejprve se vytvoří v ukázkovém zdrojovém kódu 3.24 na řádcích 1 až 2 pole o deseti prvcích, ke kterému bude následně přistupováno.

```
1 Value * len = builder.getInt32(10);
2 Value * decl = builder.CreateAlloca(Type::getInt32Ty(getGlobalContext())
      , len, "items");
3
4 Value * index = builder.getInt32(5);
5 std::vector<llvm::Value*> indexes;
6 indexes.push_back(index);
7 Value * ptr = GetElementPtrInst::CreateInBounds(decl, llvm::ArrayRef<
      llvm::Value*>(indexes), "elmptr", builder.GetInsertBlock());
```

Zdrojový kód 3.24: LLVM - pole

LLVM poskytuje instrukci GetElementPtr, která dovoluje podobně jako GCC přístup k poli na základě indexu. Instrukce se vytváří v ukázce na řádcích 4 až 7. Proměnná ptr obsahuje odkaz na daný prvek v poli, který lze pomocí Load a Store instrukcí měnit.

3.11.4 Shrnutí

GENERIC i LLVM IR poskytují podporu pro obsluhu polí, která je do značné míry podobná. V prvním kroku se pole vytvoří a ve druhém se získá odkaz na daný prvek. Po získání odkazu na prvek pole lze hodnotu číst nebo zapisovat.

3.12 Řetězce

3.12.1 Základní informace

V programu je potřeba občas reprezentovat neměnné řetězce, ve kterých jsou uloženy například zprávy pro uživatele. GCC i LLVM pro toto poskytují podporu. V dalším textu bude uvažováno vytvoření následující řetězcové konstanty.

```
1 const char text[] = "Hello world!";
```

Zdrojový kód 3.25: Řetězec

3.12.2 GCC

Vyrobení uzlu reprezentující zmíněnou konstantu je v GENERIC velmi jednoduché.

```
1 int len = strlen("Hello world!") + 1;
2 tree const_tree = build_string_literal(len, "Hello world!");
```

Zdrojový kód 3.26: GCC - řetězce

Uzel const_tree pak lze hned dále použít.

3.12.3 LLVM

U LLVM se vyrobí nejdříve globální proměnná a dále se pak vyrobí odkaz na první prvek této proměnné pomocí instrukce GetElementPtr.

```
1 Constant *constStr = ConstantDataArray::getString(getGlobalContext(), "
      Hello␣world!");
2 constStr = new GlobalVariable(*module, constStr->getType(), true,
      GlobalValue::InternalLinkage, constStr, "Hello␣world!");
3 Value * index = builder.getInt32(0);
4 Constant *indexes[] = { index, index };
5 Value * strPtr = ConstantExpr::getGetElementPtr(constStr, indexes, 2);
```

Zdrojový kód 3.27: LLVM - řetězce

3.12.4 Shrnutí

Neměnné řetězce lze generovat v obou systémech relativně jednoduše, protože je pro ně zavedena podpora.

3.13 Závěr

Přední část jazyka Mila je realizovatelná pro GCC i LLVM. Oba dva frameworky umožňují implementovat veškeré požadované funkčnosti za použití popsaného rozhraní. Ani v jednom případě nebylo třeba využívat nějakých nestandardních metod pro zajištění funkčnosti.

Co se týče obtížnosti implementace předních částí, tak bylo zjištěno, že přední část pro GCC lze implementovat snadněji. Důvodem je nízkoúrovňovost LLVM IR, kdy je třeba generovat basic bloky. Zároveň je třeba brát v úvahu existenci SSA registrů, což vede na nutnost alokovat paměť a používat Load a Store instrukce. GCC GENERIC je forma vyšší úrovně, která od výše popsaných problémů odstiňuje.

V kapitole jsou detailně popsána jednotlivá rozhraní, která existovala v době psaní práce. Je pravděpodobné, že se v budoucnosti u GCC i LLVM odehrají v rozhraních změny. Pro studium jednotlivých rozhraní se nejvíce osvědčily zdrojové kódy nebo vygenerovaná dokumentace z těchto zdrojových kódů. Tyto materiály lze doporučit každému, kdo by vyvíjel přední část pro GCC nebo LLVM.

Kapitola 4

Implementace

4.1 Úvod

Pro implementaci přední části byl využit jazyk C++ a jednotlivé její komponenty jsou pro přehlednost rozděleny do samostatných souborů, jak ukazuje tabulka 4.1. Zdrojové kódy uvedené v tabulce jsou společné pro obě přední části. Kromě společného kódu obsahuje každá přední část návštěvnickou třídu pro generování vnitřní formy překladače pro prostřední část, kterou přijímá AST. V implementaci pro GCC je tato třída v souboru gcccodegennodevisitor.cpp a v případě přední části pro LLVM se nachází v souboru llvmcodegennodevisitor.cpp.

4.2 Lexikální analyzátor

Pro účely lexikálního analyzátoru je třeba vyspecifikovat jednotlivé lexikální elementy. Tyto lexémy byly v rámci analýzy jazyka identifikovány, byly jim přiřazeny kódy a jsou popsány v tabulce 4.2.

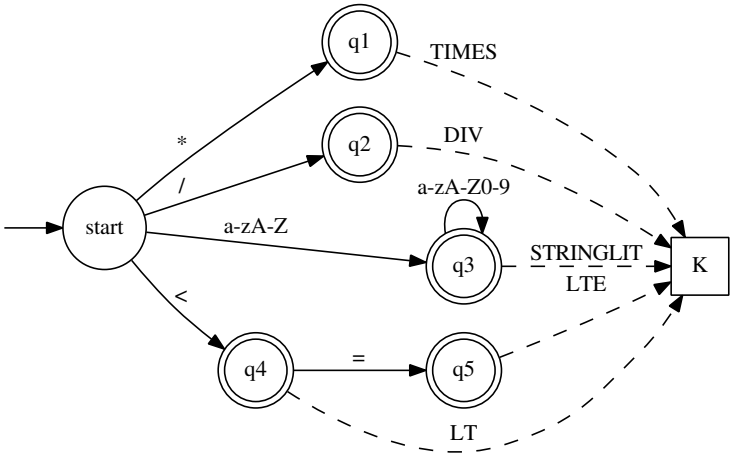
V rámci implementace byl postaven deterministický konečný automat, který rozpoznává vyjmenované lexikální elementy. Mimo těchto terminálů ale v jazyce existují ještě klíčová slova, pro které je chtěné, aby byly vraceny jako tokeny. Tato klíčová slova jsou následující: BEGIN, END, CONST, VAR, WHILE, DO, FUNCTION, INTEGER, MOD, IF, THEN, ELSE, EXIT, PROGRAM, TRUE, FALSE, AND, OR, BOOLEAN, BREAK, CONTINUE, DIV,

Tabulka 4.1: Struktura zdrojových kódů

Jméno souborů	Obsah
sfeinput.h, sfeinput.cpp	Čtení vstupních dat
sfelexan.h, sfelexan.cpp	Lexikální analyzátor
sfeparser.h, sfeparser.cpp	Parser
sfeast.h, sfeast.cpp	Abstraktní syntaktický strom

Tabulka 4.2: Lexikální analyzátor - tokeny

Kód	Význam	Kód	Význam
IDENT	Identifikátor	LPAR	(
NUM	Celé číslo	RPAR	)
PLUS	+	ASSIGN	:= (přiřazení)
MINUS	-	COMA	,
TIMES	*	SEMICOLON	;
EQ	= (rovná se)	COLON	:
NEQ	<> (nerovná se)	DOT	.
LT	<	LSQ	[
GT	>	RSQ	]
LTE	<=	STRINGLIT	Řetězecová konstanta
GTE	>=		



Obrázek 4.1: Lexikální analyzátor

PROCEDURE, ARRAY, OF, FOR, TO, DOWNTO, FORWARD. Klíčová slova jsou rozpoznávána tabulkou klíčových slov následovně: pokud je přečten token IDENT, tak lexikální analyzátor zkontroluje, jestli není přečtený identifikátor klíčové slovo. Pokud identifikátor klíčové slovo je, vrátí se token pro dané klíčové slovo, jinak se vrátí token IDENT.

Samotný konečný automat pracuje s myšlenkou, že je před čtením zdrojového kódu v počátečním stavu a jak čte jednotlivé znaky, tak se dostane do jednoho z koncových stavů. V tomto koncovém stavu je přečten lexikální element, který se vrátí a automat se zresetuje. Část takového konečného automatu ilustruje obrázek 4.1.

Platí tedy, že např. ve stavu q5, vrátí lexikální analyzátor token LTE a následně se resetuje, aby byl připraven na další symbol na vstupu. Obdobně funguje pro další vstupní znaky.

4.3 Syntaktický analyzátor

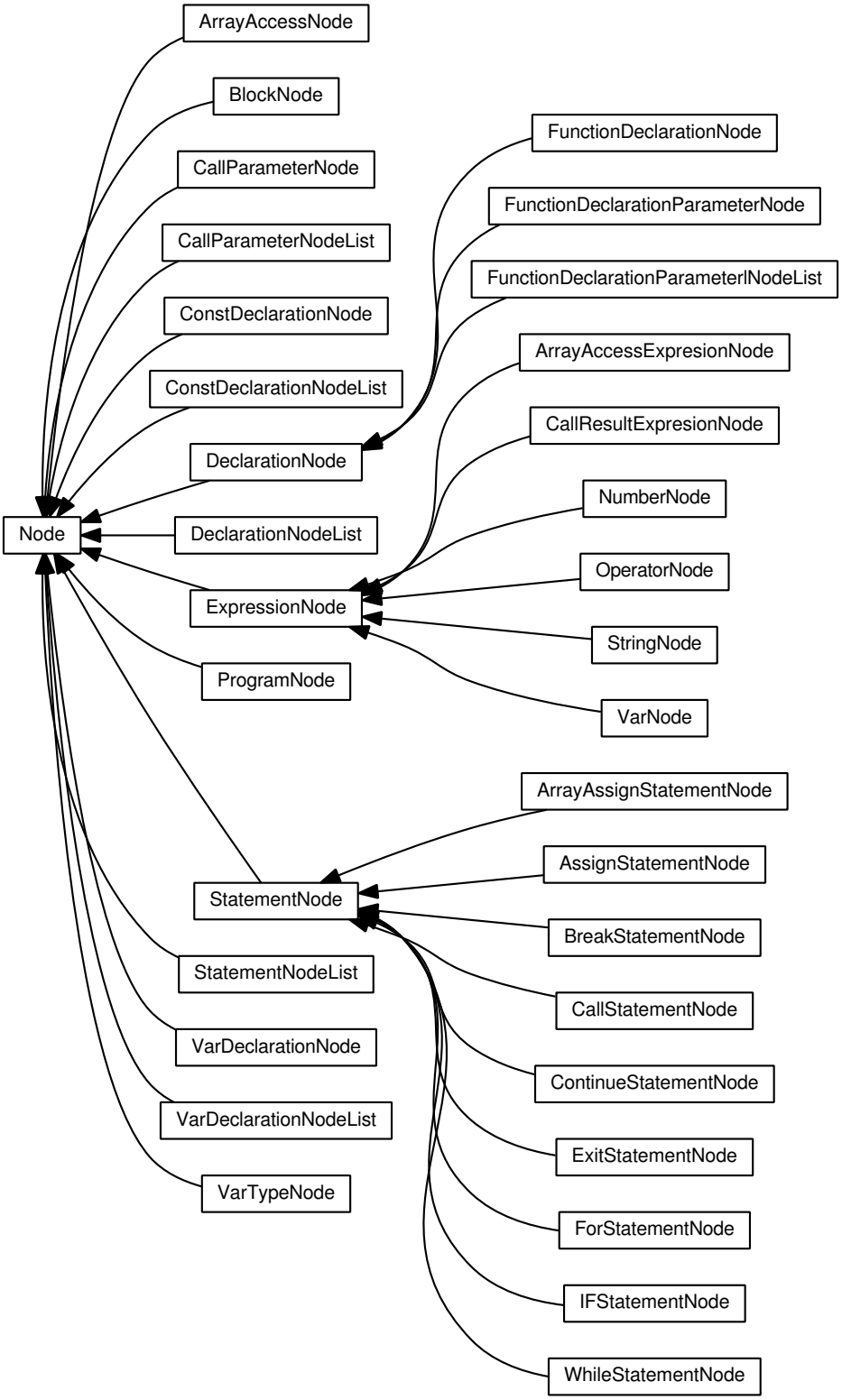
Syntaktický analyzátor, neboli parser, je sestrojen přímočaře metodou rekurzivního sestupu z LL(k) gramatiky. Kompletní gramatika jazyka Mila je popsána přílohou C. Tato gramatika byla vyrobena z EBNF zmíněné v první kapitole. Sestrojením LL(k) gramatiky bylo dokázáno, že jazyk Mila je LL(k) jazyk. Pro sestrojení parseru, který má na vstupu tokeny a na svém výstupu AST jazyka Mila, tak dále stačí sestrojit tabulku first a follow funkcí a následně vytvořit rozkladovou tabulku. Tyto kroky jsou značně mechanické a v této práci byl využit nástroj, který proces zautomatizoval. Rozkladová tabulka zde pro svou rozsáhlost není uvedena. Z rozkladové tabulky bylo metodou rekurzivního sestupu pro každý neterminál sestrojena funkce a v tělech těchto funkcí se instancují uzly AST.

4.4 Abstraktní Syntaktický strom

AST jazyka Mila byl implementován jako systém tříd vycházející ze základní třídy Node. Uzly byly implementovány tak, aby reflektovaly neterminály použité v gramatice jazyka a bylo jednoduše čitelné, co se k čemu vztahuje. Jako uzel nebyl implementován každý neterminál - buď z důvodu, že to byl pouze pomocný neterminál, aby byla gramatika LL(k), nebo že by uzel v AST nepřinesl žádnou zajímavou informaci. Obrázek 4.2 ukazuje diagram tříd AST.

4.5 Transformace AST na vnitřní formu překladače

Díky objektovému návrhu AST bylo možné implementovat vzor návštěvník. Byla vytvořena abstraktní třída NodeVisitor, která má pro každý typ uzlu metodu Visit a třídě Node byla přidána metoda Accept, která přijímá daného návštěvníka. Díky polymorfizmu pak stačilo pro GCC implementovat třídu GCCCodegenNodeVisitor a obdobně pro LLVM třídu LLVMCodegenNodeVisitor, které mají implementované jednotlivé návštěvnické metody, a generování kódu pak probíhá přijmutím návštěvnické třídy abstraktním syntaktickým stromem. Obě tyto návštěvnické třídy si ve své instanci udržují svůj kontext a lokální proměnné.



Obrázek 4.2: AST - Digram tříd

Kapitola 5

Testování

Z důvodu otestování výsledku implementace bylo vytvořeno deset testovacích programů, které musí přední části správně zpracovat. Vytvořené testovací programy jsou stavěny tak, aby pokryly všechny konstrukce jazyka Mila a co nejvíce možných kombinací těchto konstrukcí. Uvedené testovací programy byly přeloženy GCC i LLVM verzí překladače a programy nejevily žádné chyby.

Kromě nově vytvořených programů byly přeloženy i testovací programy z předcházející semestrální práce. Tyto programy mimo jiné počítaly průměr v poli prvků, maximální prvek pole, vyhodnocovaly netriviální výrazy, počítaly faktoriál - rekurzivně i cyklem, fibonaciho posloupnost, zkoumaly prvočíselnost daných čísel a byl zde i program, který faktorizoval zadané číslo. Celkově bylo takto přeloženo dalších 16 programů a bylo konstatováno, že programy fungují korektně.

Obě přední části jsou tak považovány za otestované. Zdrojové kódy těchto programů jsou přiloženy na CD. Následuje stručný popis jednotlivých nově vytvořených programů.

test1 Program provede vypsání konstanty nula na standardní výstup.

test2 V programu jsou dva podmíněné příkazy. První testuje splnění podmínky a druhý nesplnění podmínky.

test3 Testuje se for cyklus a práce s proměnnými. For cyklus v každé iteraci inkrementuje proměnnou a nakonec se pomocí podmíněného příkazu otestuje její hodnota.

test4 Program obsahuje dvě funkce počítající n-tou mocninu daného čísla. První funkce počítá mocninu rekurzivně a druhá obsahuje for cyklus. Program dále v hlavním bloku obsahuje dva vnořené for cykly, které uvnitř těla počítají pro hodnoty 0 až 20 postupně 0 až 4 mocninu definovanými funkcemi. Po získání výsledků se hodnoty porovnají, jestli jsou stejné.

test5 Deklaruje se pole s rozsahem indexů -10 až 10, které se naplní hodnotami a tyto hodnoty se vypíšou. Poté se pole seřadí alogritmem bubblesort a vypíše se výsledek.

test6 Program násobí polynomy až do výsledného stupně 1000. Polynomy jsou reprezentovány celočíselnými koeficienty uloženými v polích. Program nejprve vyzve uživatele k zadání stupně prvního polynomu a přečte od něj číslo reprezentující tento stupeň. Dále přečte koeficienty prvního polynomu a uloží je do pole. Tento postup opakuje i pro druhý polynom. Nakonec se spočítá výsledný třetí polynom a ve vhodném formátu se výsledek vypíše na obrazovku.

test7 Test simuluje zásobník pomocí pole. Ve while cyklu se program uživatele nejprve zeptá, kolik chce vložit prvků. Tyto prvky načte ze standardního vstupu a dále se program zeptá, kolik prvků chce uživatel vyjmout. Vyjmuté prvky se vypíšou na obrazovku. Program skončí ve chvíli, kdy zásobník přeteče, nebo naopak podteče.

test8 Program testuje zadávání konstant v různých formátech (dekadicky, hexadecimálně a oktalově).

test9 Program počítá největšího společného dělitele pomocí rekurze. Nejprve se uživatele zeptá na jednotlivá čísla, potom provede výpočet a nakonec vypíše výsledek na obrazovku.

test10 Program vypíše na obrazovku obdélník skládající se z čísel 0 až 9. Testují se zde zanořené for cykly.

Závěr

Práce si kladla za cíl implementovat přední části překladačů GCC a LLVM pro jazyk Mila. Implementované přední části zvládnou korektně zpracovat program v jazyce Mila a pomocí GCC a LLVM vytvořit spustitelný program. Zároveň jsou v práci rozebrány jednotlivé komponenty předních částí. Tento bod je tedy považován za splněný.

Byly úspěšně implementovány i minimální přední části, které lze využít jako šablony pro semestrální práci v předmětu PJP.

Další z cílů bylo popsat využitá rozhraní a implementace porovnat. Tomuto tématu je v práci věnována celá kapitola, kde jsou pokryty všechny potřebné struktury a metody. Porovnávací část tedy byla také splněna a lze na základě ní vytvořit novou přední část překladače.

Nakonec byly pro celkové otestování vytvořeny testovací programy. Touto sadou byly obě přední části otestovány. A lze konstatovat, že pracují korektně. Výsledné zdrojové kódy předních částí a testovacích programů jsou přiloženy na CD.

Do budoucna by šel popisovaný jazyk Mila rozšířit o další prvky - chybí zde například dynamická alokace paměti, podpora pro jiné datové typy než celé číslo nebo struktury. Ruku v ruce s rozšířením jazyka by bylo zajímavé popsat i další části rozhraní GCC a LLVM.

Literatura

[1] flex: The Fast Lexical Analyzer. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://flex.sourceforge.net>

[2] FORMAL LANGUAGES AND COMPILERS. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://brasil.cel.agh.edu.pl/ 11sustrojny/en/compiler/>

[3] GNU C Compiler Internals/GNU C Compiler Architecture. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture>

[4] LLVM Language Reference Manual. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://llvm.org/docs/LangRef.html>

[5] LLVM Programmer's Manual. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://llvm.org/docs/ProgrammersManual.html>

[6] LLVM Tutorial. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://llvm.org/docs/tutorial/>

[7] THEORY OF LANGUAGE TRANSLATION. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://nova.umuc.edu/ jarc/cmsc430/lect6.html>

[8] Free Software Foundation, I.: Bison manual. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <https://www.gnu.org/software/bison/manual/bison.html>

[9] Free Software Foundation, I.: Internals of the GNU compilers. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gccint/>

[10] Holub, J.: Přednášky k předmětu BI-AAG. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <https://edux.fit.cvut.cz/oppa/BI-AAG/prednasky/>

[11] Janoušek, J.: Prezentace k přednáškám BI-PJP. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/BI-PJP/_media/lectures/03/pjp-prednaska3.pdf>

[12] Johnson, S. C.: Yacc: Yet Another Compiler-Compiler. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://dinosaur.compilertools.net/yacc/>

[13] Kučera, J.: Programovací jazyky v historickém vývoji. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://www.fi.muni.cz/usr/jkucera/pv109/sl5.htm>

[14] Lattner, C.: The Architecture of Open Source Applications. [online], [cit. 22. 3. 2015]. Dostupné z WWW: <http://www.aosabook.org/en/llvm.html>

[15] Češka, M.: *Překladače. Skriptum.* FEI VUT Brno, 1993.

Příloha A

Seznam použitých zkratek

GCC GNU Compiler Collection

LLVM Low Level Virtual Machine

AST Abstraktní syntaktický strom

IR Intermediate language

JSA Jazyk symbolických adres

EBNF Rozvinutá Backusova-Naurova forma

SSA Static single assignment

DKA Deterministický konečný automat

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `gcc-emptyfrontend` .. zdrojové kódy minimální přední části pro GCC
  - `gcc-referencefrontend` ..... zdrojové kódy přední části pro GCC
  - `llvm-emptyfrontend` zdrojové kódy minimální přední části pro LLVM
  - `llvm-referencefrontend` ..... zdrojové kódy přední části pro LLVM
  - `test` ..... zdrojové kódy testovacích programů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF

Příloha C

LL(k) gramatika jazyka Mila

Gramatika jazyka Mila je čtveřice (N, Σ, P, S) kde N je množina neterminálních symbolů, Σ je množina terminálních symbolů, P jsou přepisovací pravidla a S je počáteční symbol.

N = {Program, Header, DeclarationList, Declaration, FunctionDeclaration, Block, FunctionDeclarationParametersList, FunctionDeclarationParameterRest, FunctionDeclarationParametersList, FunctionDeclarationParameter, FunctionDeclarationParameterRest, FunctionDeclarationParameterRest, MainBlock, Block, ConstDeclaration, ConstDeclarationList, ConstDeclarationListRest, VarDeclaration, VarDeclarationList, VarDeclarationListRest, IdentList, IdentListRest, InnerBlock, VarType, StatementList, Statement, Statement, CallStatement, IFStatement, WhileStatement, ForStatement, ExitStatememement, BreakStatememement, ContinueStatement, ArrayAssignStatement, AssignStatement, ArrayAccess, CallStatement, CallParameterList, CallParameterListRest, CallParameter, ElseStatement, Expression, ExpressionLogRest, ExpressionLog, ExpressionCmpRest, ExpressionCmp, ExpressionRest, ExpressionTerm, ExpressionTermRest, ExpressionFaktor}

Σ = {IDENT, NUM, PLUS, MINUS, TIMES, EQ, NEQ, LT, GT, LTE, GTE, LPAR, RPAR, ASSIGN, COMA, SEMICOLON, COLON, DOT, LSQ, RSQ, STRINGLIT, kwBEGIN, kwEND, kwCONST, kwVAR, kwWHILE, kwDO, kwFUNCTION, kwINTEGER, kwMOD, kwIF, kwTHEN, kwELSE, kwEXIT, kwPROGRAM, kwTRUE, kwFALSE, kwAND, kwOR, kwBOOLEAN, kwBREAK, kwCONTINUE, kwDIV, kwPROCEDURE, kwARRAY, kwOF, kwFOR, kwTO, kwDOWNTO, kwFORWARD}

S = Program

```
1  Program -> Header DeclarationList MainBlock
2  DeclarationList -> Declaration SEMICOLON DeclarationList
3  DeclarationList ->
4  Declaration -> FunctionDeclaration
5  FunctionDeclaration -> kwPROCEDURE IDENT LPAR
       FunctionDeclarationParametersList RPAR SEMICOLON Block
6  FunctionDeclaration -> kwFUNCTION IDENT LPAR
       FunctionDeclarationParametersList RPAR COLON VarType SEMICOLON Block
7  FunctionDeclarationParametersList -> FunctionDeclarationParameter
       FunctionDeclarationParameterRest
8  FunctionDeclarationParametersList ->
```

```
 9  FunctionDeclarationParameter -> IDENT COLON VarType
10  FunctionDeclarationParameterRest -> SEMICOLON
        FunctionDeclarationParameter
11  FunctionDeclarationParameterRest ->
12  Header -> kwPROGRAM IDENT SEMICOLON
13  MainBlock -> Block DOT
14  Block -> ConstDeclarationList VarDeclarationList InnerBlock
15  ConstDeclarationList ->
16  ConstDeclarationList -> kwCONST ConstDeclaration
        ConstDeclarationListRest
17  ConstDeclarationListRest -> ConstDeclarationList
18  ConstDeclarationListRest -> ConstDeclaration
19  ConstDeclaration -> IDENT EQ NUM SEMICOLON
20  VarDeclarationList -> kwVAR VarDeclaration VarDeclarationListRest
21  VarDeclarationListRest -> VarDeclarationList
22  VarDeclarationListRest -> VarDeclaration VarDeclarationListRest
23  VarDeclarationList ->
24  VarDeclaration -> IdentList COLON VarType SEMICOLON
25  IdentList -> IDENT IdentListRest
26  IdentListRest -> COMA IDENT IdentListRest
27  IdentListRest ->
28  InnerBlock -> kwBEGIN StatementList kwEND
29  InnerBlock -> Statement
30  VarType -> kwINTEGER
31  VarType -> kwBOOLEAN
32  VarType -> kwARRAY LSQ NUM DOT DOT NUM RSQ kwOF kwINTEGER
33  StatementList-> Statement SEMICOLON StatementList
34  StatementList->
35  Statement -> AssignStatement
36  Statement -> CallStatement
37  Statement -> IFStatement
38  Statement -> WhileStatement
39  Statement -> ForStatement
40  Statement -> ExitStatemement
41  Statement -> BreakStatemement
42  Statement -> ContinueStatement
43  Statement -> ArrayAssignStatement
44  AssignStatement -> IDENT ASSIGN Expression
45  ArrayAssignStatement -> ArrayAccess ASSIGN Expression
46  ArrayAccess -> IDENT LSQ Expression RSQ
47  CallStatement -> IDENT LPAR CallParameterList RPAR
48  CallParameterList -> CallParameter CallParameterListRest
49  CallParameterList ->
50  CallParameterListRest -> COMA CallParameter CallParameterListRest
51  CallParameterListRest ->
52  CallParameter -> Expression
53  CallParameter -> STRINGLIT
54  IFStatement -> kwIF Expression kwTHEN InnerBlock ElseStatement
55  ElseStatement -> kwELSE InnerBlock
56  ElseStatement ->
57  WhileStatement -> kwWHILE Expression kwDO InnerBlock
58  ForStatement -> kwFOR IDENT ASSIGN Expression kwTO Expression kwDO
        InnerBlock
59  ForStatement -> kwFOR IDENT ASSIGN Expression kwDOWNTO Expression kwDO
        InnerBlock
60  Expression -> ExpressionLog ExpressionLogRest
61  ExpressionLogRest -> kwAND ExpressionLog ExpressionLogRest
62  ExpressionLogRest -> kwOR ExpressionLog ExpressionLogRest
63  ExpressionLogRest ->
64  ExpressionLog -> ExpressionCmp ExpressionCmpRest
65  ExpressionCmpRest -> EQ ExpressionCmp ExpressionCmpRest
66  ExpressionCmpRest -> NEQ ExpressionCmp ExpressionCmpRest
```

```
67  ExpressionCmpRest -> LT ExpressionCmp ExpressionCmpRest
68  ExpressionCmpRest -> LTE ExpressionCmp ExpressionCmpRest
69  ExpressionCmpRest -> GT ExpressionCmp ExpressionCmpRest
70  ExpressionCmpRest -> GTE ExpressionCmp ExpressionCmpRest
71  ExpressionCmpRest ->
72  ExpressionCmp -> ExpressionTerm ExpressionRest
73  ExpressionRest -> PLUS ExpressionTerm ExpressionRest
74  ExpressionRest -> MINUS ExpressionTerm ExpressionRest
75  ExpressionRest ->
76  ExpressionTerm -> ExpressionFaktor ExpressionTermRest
77  ExpressionTermRest -> TIMES ExpressionFaktor ExpressionTermRest
78  ExpressionTermRest -> kwDIV ExpressionFaktor ExpressionTermRest
79  ExpressionTermRest -> kwMOD ExpressionFaktor ExpressionTermRest
80  ExpressionTermRest ->
81  ExpressionFaktor -> IDENT
82  ExpressionFaktor -> CallStatement
83  ExpressionFaktor -> ArrayAccess
84  ExpressionFaktor -> NUM
85  ExpressionFaktor -> kwTRUE
86  ExpressionFaktor -> kwFALSE
87  ExpressionFaktor -> LPAR Expression RPAR
```