TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Automatizace překladu mezi verzemi jazyka ABAP

Diplomová práce

Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie

Autor práce: **Bc. Jan Vacek**
Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2020

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

The automatization of inter-version ABAP language compilation

Diploma thesis

Study programme: N2612 Electrical Engineering and Informatics
Study branch: Information Technology

Author: **Bc. Jan Vacek**
Supervisor: Ing. Lenka Kosková Třísková, Ph.D.
Institute of New Technologies and Applied Informatics

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Automatizace překladu mezi verzemi jazyka ABAP

Jméno a příjmení: **Bc. Jan Vacek**
Osobní číslo: M17000141
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Seznamte se s metodami automatizace překladu mezi programovacími jazyky a současnými nástroji pro tvorbu překladačů.
2. Seznamte se s prostředím SAP, skriptovacím jazykem ABAP a jeho verzemi.
3. Popište rozdíly mezi starou a novou verzí jazyka ABAP (7.4 a starší verze), nejlépe formou srovnání gramatik obou jazyků.
4. Navrhněte vhodný způsob překladu mezi novou a starou syntaxí jazyka ABAP.
5. Navržený postup realizujte s pomocí vybraných nástrojů.

Rozsah grafických prací: dle potřeby
Rozsah pracovní zprávy: 40-50 stran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

[1] Čejka M., Hruška T., Beneš M.: Překladače, Učební text VUT Brno.
[2] Wirth N.: Compiler Construction, Addison-Wesley, 1996, ISBN 0-201-40353-6.
[3] Parr T.: The Definitive ANTLR Reference, The Pragmatic Bookshelf, 2007, ISBN: 0-9787392-5-6.
[4] Bandari K.: Complete ABAP, SAP Press, 2017, ISBN 978-1-4932-1273-6.

Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Datum zadání práce: 9. října 2019
Předpokládaný termín odevzdání: 18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 17. října 2019

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že texty tištěné verze práce a elektronické verze práce vložené do IS/STAG se shodují.

1. února 2020 Bc. Jan Vacek

Poděkování

Na tomto místě bych chtěl poděkovat vedoucí práce paní Ing. Lence Koskové Třískové Ph.D. za vedení této diplomové práce, věcné připomínky a cenné rady. Dále bych chtěl poděkovat společnosti T-MC66, s.r.o., zejména panu Ing. Lukáši Sýkorovi za umožnění realizace této diplomové práce a panu Ing. Pavlu Kaiserovi za užitečné rady a informace. Velké díky patří také rodičům, kteří mě během celého studia podporovali a stáli při mně.

Abstrakt

Diplomová práce je zaměřena na automatizaci překladu mezi verzemi jazyka ABAP, který je používán k vývoji v systému SAP. Zabývá se popisem a návrhem řešení problematiky zpětné kompatibility mezi verzemi jazyka ABAP. Práce je dále zaměřena na návrh gramatiky a následný návrh a realizaci vhodného způsobu překladu mezi jednotlivými verzemi jazyka ABAP.

Navržená gramatika je zapsána dle normy EBNF. Překladač je realizován v jazyce Java s využitím nástroje pro tvorbu překladačů ANTLR ve verzi 4. Práce popisuje i alternativní řešení převodu mezi jednotlivými verzemi jazyka ABAP, které je zapsané v jazyce ABAP a spustitelné přímo v systému SAP.

Klíčová slova

systém SAP, ABAP, překladač, gramatika, ANTLR

Abstract

Diploma thesis is focused on automation of translation between versions of ABAP language, which is used for developement in SAP system. The thesis deals with description and design of problems with backwards compatibility between versions of ABAP language. The thesis also focuses on the design of the grammar and the resulting design and implementation of a suitable method of translation between different versions of the ABAP language.

The designed grammar is written according to EBNF standard. The compiler is implemented in Java with the use of the ANTLR compiler creation tool in version 4. The thesis also describes an alternative solution of conversion between individual versions of ABAP language, which is written in ABAP language and executable directly in the SAP system.

Keywords

SAP system, ABAP, compiler, grammar, ANTLR

Obsah

1 Úvod 13

1.1 Motivace 13

1.2 Cíle práce 13

2 Překladače 15

2.1 Gramatika jazyka 15

2.2 Lexikální analýza 16

2.3 Syntaktická analýza 17

2.4 Tabulka symbolů 18

2.5 Sémantická analýza 18

2.6 Generátor kódu 19

3 Nástroje pro tvorbu překladačů 20

3.1 ANTLR 20

3.2 JavaCC 20

3.3 Lex 21

3.4 Yacc 21

3.5 Bison 21

4 Systém SAP 22

4.1 Systém ERP 22

4.2 Systém SAP ERP 23

4.3 Historie systému SAP 23

4.4 Architektura systému SAP 24

4.5 Programovací jazyk ABAP 26

4.5.1 Základní typy aplikací 26

4.6 Verze jazyka ABAP 27

4.6.1 Změny ve verzi 7.4 a porovnání se starší verzí 31

5 Gramatika jazyka ABAP 46
5.1 Pravidla gramatiky a jejich popis 46
5.1.1 Počáteční pravidlo gramatiky 46
5.1.2 Příkaz jazyka ABAP 46
5.1.3 Operandy 47
5.1.4 Operátory 48
5.1.5 Datové typy 50
5.1.6 Deklarace proměnných 51
5.1.7 Přiřazení hodnoty proměnné 51
5.1.8 Hodnota proměnné 52
5.1.9 Smyčka LOOP 52
5.1.10 Volání metod 53
5.1.11 Čtení dat z interní tabulky 53
5.1.12 Čtení dat z databázové tabulky 53
5.1.13 Příkaz IF 54
5.1.14 Příkaz CASE 54
6 Návrh překladače 56
6.1 Úvod do problematiky 56
6.2 Návrh řešení 56
7 Implementace překladače 58
7.1 Implementace s využitím nástrojů pro tvorbu překladačů 58
7.1.1 Výběr nástroje pro realizaci 58
7.1.2 Realizace řešení s vybraným nástrojem 58
7.2 Implementace v systému SAP 64
7.2.1 Vstup a výstup programu 65
7.2.2 Tabulky s pravidly 66
7.2.3 Princip převodu 67

8 Závěr 71

9 Seznam použité literatury.. 72

10 Přílohy 73

Příloha A – zdrojový kód Visitoru překladače... 73

Seznam obrázků

Obrázek 1 – Schéma systému non-ERP. Překresleno a přeloženo z [4]........................ 22
Obrázek 2 - Schéma systému ERP. Překresleno a přeloženo z [4]............................... 22
Obrázek 3 - Schéma komunikace v systému SAP. Překresleno a přeloženo z [4]. 25
Obrázek 4 - Schéma principu funkce reportu. Překlesleno a přeloženo z [4]. 26
Obrázek 5 - Schéma překladače... 59
Obrázek 6 – Ukázka derivačního stromu pravidla.. 60
Obrázek 7 - Grafické rozhraní překladače. 61
Obrázek 8 - Hlášení o neúspěšném překladu. 61
Obrázek 9 - Schéma převodního programu. 65
Obrázek 10 - Grafické rozhraní programu v systému SAP. 65
Obrázek 11 - Výstup programu v systému SAP. 66
Obrázek 12 - schéma tabulky pro uchování převodových pravidel.............................. 66
Obrázek 13 - Upozornění na neaktivní program. 68

Seznam tabulek

Tabulka 1 - Evoluce systému SAP. Převzato z [4]. 24
Tabulka 2 - Vývoj verzí jazyka ABAP. Převzato a přeloženo z [13]. 28
Tabulka 3 - Tabulka změn - Inline deklarace. Převzato z [8]. 31
Tabulka 4 - Tabulka změn - Práce s tabulkami. Převzato z [8]. 32
Tabulka 5 - Tabulka změn - CONV. Převzato z [8]. 34
Tabulka 6 - Tabulka změn - FOR. Převzato z [8]. 35
Tabulka 7 - Tabulka změn - DO-WHILE. Převzato z [8]. 36
Tabulka 8 - Tabulka změn – REDUCE 1. Převzato z [8]. 36
Tabulka 9 - Tabulka změn - REDUCE 2. Převzato z [8]. 37
Tabulka 10 - Tabulka změn - CORRESPODING. Převzato z [8]. 38
Tabulka 11 - Tabulka změn - Textové řetězce 1. Převzato z [8]. 40
Tabulka 12 - tabulka změn - Textové řetězce 1. Převzato z [8] 40
Tabulka 13 - Tabulka změn - Třídy a metody 1. Převzato z [8]. 42
Tabulka 14 - Tabulka změn - Třídy a metody 2. Převzato z [8]. 42
Tabulka 15 - Tabulka změn - Třídy a metody 3. Převzato z [8]. 43
Tabulka 16 - Tabulka změn - Třídy a metody 4. Převzato z [8]. 43
Tabulka 17 - Příklad převodového pravidla. 67

Seznam zdrojových kódů

Zdrojový kód 1 - Přehled změn - VALUE. Převzato z [8] 35
Zdrojový kód 2 - Přehled změn - COND. Převzato z[8]. 37
Zdrojový kód 3 - Přehled změn - SWITCH. Převzato z [8]. 38
Zdrojový kód 4 - Přehled změn - Textové řetězce. Převzato z [8]. 39
Zdrojový kód 5 - Přehled změn - GROOP & LOOP 1. Převzato z [8]. 41
Zdrojový kód 6 - Přehled změn - GROUP BY & LOOP 2. Převzato z [8]. 42
Zdrojový kód 7 - Přehled změn - Mesh 1. Převzato z [8]. 44
Zdrojový kód 8 - Přehled změn - Mesh 2. Převzato z [8]. 44
Zdrojový kód 9 - Přehled změn - Filtry. Převzato z [8]. 45
Zdrojový kód 10 - Třída pravidla Declaration. 62
Zdrojový kód 11 - Metoda Visitoru – visitDecalariotn. 63
Zdrojový kód 12 - Metoda get_children_list. 64
Zdrojový kód 13 - Příklad převedeného řádku. 69
Zdrojový kód 14 - Uložení převedeného programu. 69

Seznam zkratek

SAP – Systeme, Anwendungen, Produkte in der Datenverarbeitung

ERP – Enterprise Resource Planning

ABAP – Advanced Business Application Programming

EBNF – Extended Backus-Naur Form

FI - SAP Financial Accounting

SD – SAP Sales and Distribution

MM – SAP Materials Management

HR – SAP Human Resource

WM – SAP Warehouse Management

PP – SAP Production Planning

1 Úvod

1.1 Motivace

Toto téma vzniklo ve společnosti T-MC66, která se zabývá nastavením a vývojem v systému SAP na základě potřeb konkrétního zákazníka. Nejvýznamnějším zákazníkem je společnost Škoda Auto a.s.

Hlavní motivací je využitelnost výsledného programu v praxi. Přínos spočívá především v urychlení převodu jednotlivých programů a modulů mezi verzemi jazyka ABAP, který doposud probíhal manuálně.

1.2 Cíle práce

Hlavním cílem práce je navrhnout vhodný způsob překladu mezi novou a starou verzí syntaxe programovacího jazyka ABAP. Následně tento návrh realizovat s pomocí vybraných nástrojů.

Aby bylo možné způsob překladu navrhnout, je nezbytné se seznámit s metodami automatizace překladu mezi programovacími jazyky a současnými nástroji pro tvorbu překladačů. Následně je velmi důležité se seznámit s prostředím systému SAP a programovacím jazykem ABAP a jeho verzemi.

Podmínkou realizace překladu mezi jednotlivými verzemi jazyka ABAP, je nutné popsat vzájemné rozdíly. A to nejlépe srovnáním gramatik obou verzí jazyka. Výsledný překladač by měl tedy, s využitím gramatik obou jazyků, umožňovat překlad mezi novou a starou verzí jazyka ABAP. Přičemž by měl uživateli poskytovat zpětnou vazbu o úspěšnosti překladu.

Tuto práci lze rozdělit na dvě hlavní části, rešeršní část a implementační část. Rešeršní část je zaměřena na teorii překladačů, přibližuje princip jejich funkce a obsahuje na popis současných nástrojů pro jejich tvorbu. Rešeršní část dále obsahuje základní popis systému SAP, včetně shrnutí jeho historie a vývoje. V této části je také popsán jazyk ABAP a jeho verze.

V druhé části je podrobně popsána gramatika nové verze jazyka ABAP, včetně srovnání rozdílů s gramatikou staré verze. Následně je zde diskutováno, jak rozdíl ve verzích ovlivní automatický převod a je předložen návrh možného řešení popsaného problému. V posledním bloku praktické části je popsané navržené řešení problému, tj. automatizace překladu mezi verzemi jazyka ABAP.

2 Překladače

Překladač je program, jehož funkcí je čtení zdrojového programu a následný převod do ekvivalentního cílového programu. Zdrojový program je napsaný ve zdrojovém jazyce, cílový program v cílovém jazyce. Je důležité, aby překladač během překladu informoval uživatele o případných chybách ve zdrojovém programu. Chybami se rozumí například překlepy v názvech datových typů, chybějící závorky, středníky a podobně. Není-li uvedeno jinak, vychází veškeré informace v této kapitole z [1].

2.1 Gramatika jazyka

Gramatika jazyka představuje souhrn pravidel, které definují jazyk. Obecně se gramatika jazyka skládá z terminálních a neterminálních symbolů, startovacího symbolu a produkčních pravidel. Informace o gramatikách jsou čerpány z [5].

Terminály jsou základní stavební kameny gramatiky. Terminály mohou být jednotlivé znaky nebo číslice, ze kterých se následně tvoří textové řetězce nebo číslice reprezentující názvy proměnných nebo jejich hodnoty. Za terminály se také považují klíčová slova daného programovacího jazyka, například *if*, *else* a podobně. Neterminály jsou syntaktické proměnné označující množinu textových řetězců. Tato množina pomáhá definovat jazyk generovaný gramatikou.

Neterminály také slouží k uložení hierarchické struktury jazyka, která je užitečná pro syntaktickou analýzu a následný překlad. Jeden neterminální symbol je rozlišován jako startovací symbol gramatiky. Množina řetězců, kterou lze z tohoto symbolu s pomocí pravidel odvodit, je jazyk definovaný gramatikou.

Produkční pravidla gramatiky specifikují způsob, kterým jsou terminální a neterminální symboly kombinovány do textových řetězců. V textu se zabývám pouze bezkontextovými gramatikami, pro něž platí, že levá strana produkčního pravidla obsahuje vždy jeden neterminál a pravá strana kombinaci terminálů a neterminálů.

2.2 Lexikální analýza

Lexikální analýza je první fází překladu, jejíž hlavním úkolem je číst znaky zdrojového programu, které jsou následně seskupovány do posloupnosti lexikálních symbolů. Symboly v lexikální posloupnosti představují logicky související posloupnosti znaků, například identifikátory proměnných, operátory matematických operací a podobně. Tyto celky se nazývají lexémy.

Lexémy jsou dále předány syntaktickému analyzátoru. Během lexikální analýzy mohou být prováděny další úkoly. Jedním z těchto úkolů je odstraňování uživatelských komentářů a odsazení (mezery, tabulátory a konce řádků) ze zdrojového programu. Dále může lexikální analyzátor sledovat znaky reprezentující konce řádků. Díky této informaci je možné ke každému chybovému hlášení připojit číslo řádku, na kterém se daná chyba vyskytla.

V běžné praxi je lexikální analyzátor oddělený od toho syntaktického. Tento přístup má několik důvodů. Prvním a asi nejpodstatnějším důvodem je zjednodušení návrhu překladače. Oddělení lexikální a syntaktické analýzy umožňuje jednu nebo obě části překladače zjednodušit, například syntaktický analyzátor obsahující pravidla pro uživatelské komentáře a mezery je podstatně složitější než analyzátor, který počítá s tím, že tyto objekty byly již odstraněny lexikálním analyzátorem. Během lexikální analýzy může dojít k zastavení překladu při nalezení chyby. Chyby, které dokáže lexikální analyzátor odhalit, jsou zejména překlepy v textu.

Lexikální analyzátor je schopen se z těchto chyb zotavit. Existuje více přístupů zotavení. Nejsnadnější metodou zotavení je cyklické vynechávání symbolů, dokud se lexikálnímu analyzátoru nepodaří rozpoznat další správně vytvořený symbol. Další možností je vytvoření speciálního terminálního symbolu, který není obsažen v gramatice jazyka. V tomto případě je detekce chyby a její náprava přenechána na syntaktický analyzátor. Oba přístupy jsou běžně využívány v praxi a ve většině případů jsou dostatečně účinné. Dalšími možnými způsoby zotavení z chyb jsou vypuštění přebývajícího znaku, vložení chybějícího znaku, náhrada nesprávného znaku správným nebo vzájemná výměna dvou sousedních znaků. Výstupem lexikálního analyzátoru je kromě zdrojového kódu ve formě textového proudu také tabulka symbolů, se kterou dále pracuje syntaktický a sémantický analyzátor.

2.3 Syntaktická analýza

Účelem syntaktické analýzy je zjistit, zda zdrojový text tvoří větu odpovídající gramatice překládaného jazyka. K tomu je využívána posloupnost lexikálních symbolů neboli lexémů. Posloupnost lexémů je výstupem lexikální analýzy. Pokud se ve zdrojovém textu vyskytne chyba, překladač ji nahlásí a obvykle se pokusí z dané chyby nějakým způsobem zotavit a pokračovat dále v překladu, případně odhalit další chyby.

Během implementace překladače je obvykle využit jeden ze dvou základních přístupů. Překlad shora dolů nebo zdola nahoru. Názvy odpovídají směru vytváření derivačního stromu.

Pokud překladač překládá shora dolů, vychází ze startovacího symbolu gramatiky a postupným rozšiřováním neterminálních symbolů se snaží dospět až symbolům terminálním, které odpovídají posloupnosti lexémů na vstupu. Při překladu zdola nahoru se překladač snaží postupně redukovat lexémy až na startovací symbol gramatiky. Oběma přístupům odpovídají dva typy gramatik, LL a LR gramatiky.

Dalším důležitým úkolem syntaktického analyzátoru je diagnostická činnost, jejímž cílem je během jednoho průchodu zdrojovým programem odhalit co nejvíce chyb. Aby byl syntaktický analyzátor tohoto schopen, je nutné implementovat prostředky, které analyzátoru umožní pokračovat v kontrole kódu i po výskytu syntaktické chyby.

Pro detekci chyb jsou využívána jednotlivá pravidla gramatiky jazyka. Chyby, které je syntaktický analyzátor schopen detekovat jsou například chybějící závorky v matematickém výrazu, chybějící středníky a podobně. Zotavení se ze syntaktické chyby není obecně jednoduchým úkonem. Obecný postup, který je běžně používán v metodách pro zotavení se skládá z několika kroků.

Prvním krokem je, po odhalení syntaktické chyby, nalezení místa bodu zotavení. Jedná se o místo v kódu, odkud může syntaktický analyzátor pokračovat v činnosti, čímž dojde k vynechání určité části textu. Tento bod je obvykle dán nalezením symbolu z množiny klíčových slov. V dalším kroku provede syntaktický analyzátor synchronizaci podle pozice nalezeného klíče v gramatice a pokračuje v činnosti. Je důležité, aby množina klíčů byla definována tak, že jejich výskyt je v gramatice jednoznačný. Tím lze zajistit vyšší spolehlivost syntaktického analyzátoru při zotavení.

2.4 Tabulka symbolů

Tabulka symbolů je objekt generovaný lexikálním analyzátorem a slouží k uchování informací o pojmenovaných objektech. Uchované objekty mohou být deklarovány explicitně nebo implicitně.

Mezi explicitně deklarované objekty patří uživatelské datové typy, proměnné procedury a další. Implicitně deklarovanými objekty se rozumí standardní datové typy, procedury, funkce, pomocné proměnné deklarované překladačem atd.

Informace uložené v tabulce symbolů jsou využívány zejména k popisu vztahu mezi deklarací a použitím objektu, které nelze popsat bezkontextovou gramatikou. Tabulka symbolů může být dále využívána k provádění typové kontroly a také ke generování mezikódu nebo cílového kódu.

2.5 Sémantická analýza

Sémantická analýza zpracovává syntaktický strom získaný ze syntaktické analýzy. Hlavním úkolem sémantického analyzátoru je přiřazení významu jednotlivým operacím a příkazům. Velmi důležitou součástí sémantické analýzy je typová kontrola. V rámci typové neboli statické kontroly jsou detekovány různé druhy chyb.

Prvním příkladem je typová kontrola, která detekuje použití operátoru na vzájemně nekompatibilní datové typy. Například pokud je aplikován operátor pro sčítání na proměnnou typu pole a typu funkce.

Dalším příkladem statické kontroly je kontrola toku řízení. Například použití klíčového slova *break* v jazyce C opustí nejmenší obklopující příkaz *while*, *for* nebo *switch*. V případě, že žádný takový obklopující příkaz neexistuje, nastane chyba.

Další typ statické kontroly je kontrola jedinečnosti, to znamená, že se ve zdrojovém kódu nesmí vyskytovat deklarace proměnné se stejným názvem více než jednou.

2.6 Generátor kódu

Překladač může obsahovat dva typy generátorů, generátor mezikódu a generátor cílového kódu. Program zapsaný v mezikódu většinou slouží pro virtuální běhové prostředí, které se vyskytuje zejména v interpretovaných jazycích, jako je například Java. Mezikód se dále používá k případným optimalizacím před vygenerováním cílového kódu v cílovém jazyce.

Během optimalizace se překladač pokouší o vylepšení mezikódu. Úspěšnost optimalizace je posuzována podle základních kritérií, kterými jsou délka generovaného cílového programu, rychlost výpočtu a paměťová náročnost. Pojem optimalizace tedy vždy neznamená nalezení nejlepší varianty.

Poslední fází překladu je generování cílového kódu. Generátor cílového programu zpracovává posloupnost příkazů ve vnitřním jazyku překladače, na základě kterých je vygenerován program zapsaný v cílovém jazyce. Vygenerovaný program je sémanticky ekvivalentní se zdrojovým programem. Ve většině případů je výstupem generátoru cílového kódu posloupnost strojových instrukcí s absolutními nebo relativními adresami, případně posloupnost příkazů jazyka symbolických instrukcí.

3 Nástroje pro tvorbu překladačů

Nástroje pro tvorbu překladačů mohou využívat moderních vývojových prostředí, která obsahují užitečné nástroje jako například verzovací systémy, nástroje pro ladění zdrojového kódu a podobně. Většina nástrojů pro tvorbu překladačů poskytuje vysokou míru abstrakce. To znamená, že detaily implementace jsou zvenčí nepřístupné. Nejpoužívanější nástroje pro tvorbu překladačů obsahují: parser generátory, scanner generátory, nástroje pro syntaxí řízený překlad, generátory kódů.

Úkolem parser generátoru je automaticky vygenerovat syntaktický analyzátor na základě gramatiky jazyka. Cílem scanner generátoru je vygenerovat lexikální analyzátor z klíčových slov jazyka popsaných regulárními výrazy. Nástroj pro syntaxí řízený překlad generuje skupinu postupů pro průchod syntaktickým stromem a generování mezikódu. Generátor kódu obstarává překlad jednotlivých operací z jazyka mezikódu do cílového jazyka. Uvedené informace vychází z [6].

V této kapitole je popsáno několik vybraných nástrojů pro tvorbu překladačů. Ke každému nástroji jsou uvedené základní informace o tom, jaké jazyky daný nástroj podporuje pro implementaci překladače a které výše zmíněné komponenty obsahuje.

3.1 ANTLR

Another Tool For Language Recogniton, zkráceně ANTLR obsahuje parser generátor, scanner generátor a nástroje pro řízený překlad. ANTLR je kompletně napsaný v jazyce Java. Překladač jako takový může být generován v několika jazycích. Mezi ně patří Java, C#, C++, Python a několik dalších. Zpracovávaný jazyk je definován bezkontextovou gramatikou zapsanou v EBNF. Nástroj ANTLR je dostupný ve formě doplňkových modulů, plug-inů, pro vývojová prostředí IntelliJ, NetBeans, Eclipse, Visual Studio Code a další. Uvedené informace vychází z [3].

3.2 JavaCC

Stejně jako v případě ANTLR jsou součástí JavaCC parser generátor, scanner generátor a nástroje pro syntaxí řízený překlad. JavaCC je čistě napsané v jazyce Java, díky čemuž

je možné jej spustit na všech platformách kompatibilních s jazykem Java ve verzi 1.1 nebo novější. Zdrojový kód je zpracováván a překládán využitím přístupu shora dolů. JavaCC akceptuje gramatiky zapsané v EBNF. Lexikální specifikace, jako jsou regulární výrazy, řetězce atd. i pravidla gramatiky jsou zapsána v jednom souboru. Takto uložená gramatika je lépe čitelná a udržovatelná. Chybová hlášení JavaCC patří k těm nejlepším mezi parser generátory. Parser vygenerovaný pomocí JavaCC je schopen jasně určit, kde v překladu došlo k chybě a poskytnout kompletní diagnostické hlášení. Překladač implementovaný za pomoci JavaCC je napsaný v jazyce Java. Uvedené informace vychází z [10].

3.3 Lex

Na rozdíl od ANTLR nebo JavaCC slouží Lex pouze pro generování lexikálních analyzátorů. Lex je často využíván spolu s generátorem syntaktických analyzátorů, například yacc nebo Bison. Pro rozpoznávání lexémů využívá Lex regulární výrazy. Vstupem je soubor s příponou .l popisující generování lexikálního analyzátoru. Ten je překladačem přeložen na program napsaný v jazyce C, který je předán dále. Existuje také opensource varianta pod označením flex. Uvedené informace vychází z [12].

3.4 Yacc

Yacc, neboli Yet Another Compiler-Compiler obsahuje pouze parser generátor. Pro plnohodnotnou syntaktickou analýzu je potřeba použít lexikální analyzátor, jako například Lex. Vstupem je gramatika jazyka doplněná o útržky kódu v jazyce C, označované jako akce. Každá akce je přiřazená nějakému pravidlu gramatiky. Výstupem je parser v jazyce C, který vykonává akce přiřazené pravidlům v okamžiku, kdy je dané pravidlo rozpoznáno. Uvedené informace vychází z [12].

3.5 Bison

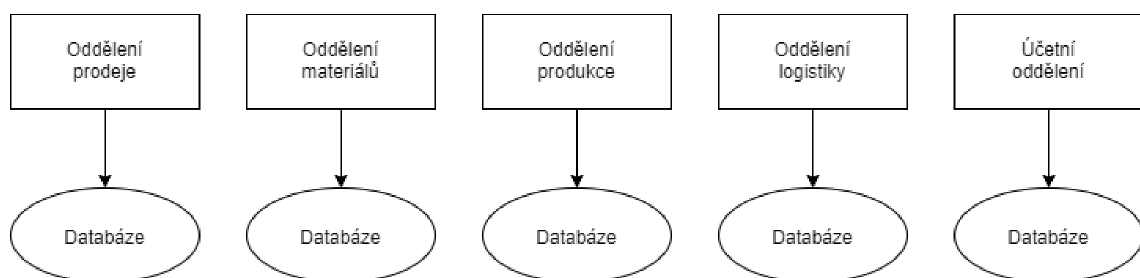
Stejně jako yacc obsahuje pouze parser generátor. Velmi často bývá kombinován s Flex, který má na starost lexikální analýzu. Vygenerovaný parser je napsaný buď v jazyce C, C++ nebo Java. Uvedené informace vychází z [11].

4 Systém SAP

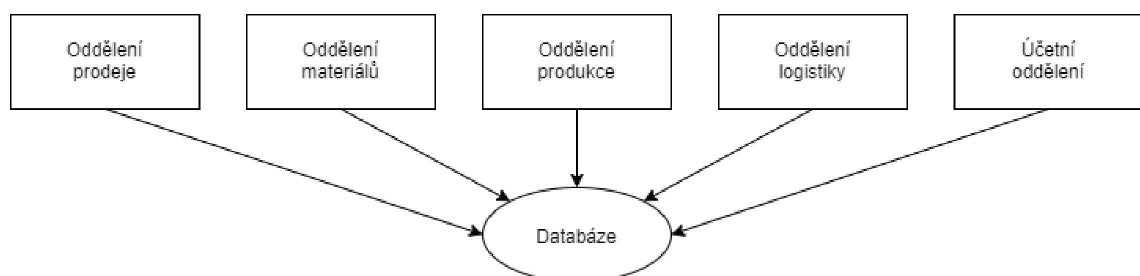
SAP poskytuje software pro plánování podnikových zdrojů (ERP – Enterprise Resource Planning). Programovací jazyk ABAP je používán pro vývoj aplikací běžících v systému SAP. Pochopení principu funkce systému ERP a seznámení se se systémem SAP jako takovým je nezbytné pro porozumění programování v jazyce ABAP. Není-li uvedeno jinak, veškeré informace vychází z knihy Complete ABAP [4].

4.1 Systém ERP

ERP software pro správu chodu firmy integruje různorodé obchodní činnosti. Takový systém zpravidla obstarává společnou databázi, ve které se nachází všechna data z různých odvětví společnosti. Tento datový model umožňuje přístup k datům napříč všemi odděleními. Rozdíl mezi ERP a non-ERP systémem spočívá právě v datovém modelu. V non-ERP systému zachyceném na Obrázku 1 je každému oddělení k dispozici vlastní databáze pro uchovávání dat. Tento přístup značně komplikuje získávání dat z jiných oddělení, zejména pak udržování aktuálních a korektních dat napříč databázemi. Na opačné straně, ERP systém (viz Obrázek 2) využívá jednu společnou databázi pro všechny oddělení firmy. Každý ERP systém poskytuje mnoho bezpečnostních opatření, aby byl zamezen neoprávněný přístup k datům.



Obrázek 1 – Schéma systému non-ERP. Překresleno a přeloženo z [4].



Obrázek 2 - Schéma systému ERP. Překresleno a přeloženo z [4].

4.2 Systém SAP ERP

SAP je velice rozšířený, zejména díky obrovské flexibilitě a snadnému přizpůsobení. Systém SAP je rozdělen do modulů, každý modul je vázán na konkrétní obchodní proces obchodní jednotky. Moduly se dělí na funkční moduly a technické moduly. Funkční moduly mají na starost obchodní procesy a každodenní aktivity společnosti, jako například účetnictví, plánování výroby a mnoho dalšího. Technické moduly pokrývají technickou stránku systému SAP a jsou potřebné pro přizpůsobení systému potřebám konkrétní společnosti. Mezi technické moduly patří i programovací jazyk ABAP.

4.3 Historie systému SAP

První verze systému SAP s označením R/1 byla představena v roce 1972. Tento systém sloužil ke správě finančního účetnictví a v tehdejší době neobsahoval prezentační, aplikační nebo databázovou vrstvu, které nalezneme v současné verzi systému. Později v sedmdesátých letech byla představena nová verze systému R/2, s dvouvrstvou architekturou. Tato architektura obsahovala prezentační vrstvu a aplikačně-databázovou vrstvu. S vývojem technologií v devadesátých letech došlo k přechodu na třívrstvou architekturu pod označením R/3. Stejnou architekturu, skládající se z prezentační, aplikační a databázové vrstvy, využívají i aktuální verze systému SAP. V roce 2002 s příchodem verze SAP R/3 4.7 byla poprvé představena technologická platforma SAP Web Application Server, která obsahovala Business Server Pages aplikace a J2EE. V roce 2006 byl vydán první balíček nazvaný Enhancement Package. Tyto balíčky poskytují dodatečné funkce zákazníkům, bez toho, aniž by zasahovaly do jádra celého systému. Zákazníci si mohou aktivovat pouze ty nové funkce, které využijí, a to pomocí frameworku Switch. Nové balíčky jsou vydávány pravidelně, nejčastěji každé dva roky. Nové funkce obsažené v balíčcích jsou určeny jak pro obecné, tak i pro průmyslově specifické procesy.

Tabulka 1 - Evoluce systému SAP. Převzato z [4].

Rok	Verze
1972	SAP R/1 (R – real-time; 1 – jednovrstvá architektura)
Konec 70. let	SAP R/2 (R – real-time; 2 – dvouvrstvá architektura)
1992	SAP R/3 (R – real-time; 3 – třívrstvá architektura)
2001	Představení SAP Web Application Server (technologická platforma)
2002	SAP R/3 4.70 (SAP Web Application Server 6.20)
2003	mySAP ERP 2003 (SAP Web Application Server 6.30)
2004	mySAP ERP 2004 ECC 5.0 (SAP Web Application Server 6.40; NetWeaver 2004)
2005	mySAP ERP 2005 ECC 6.0 ERP 6.0 (SAP Web Application Server 7.00; NetWeaver 2004s/7)
2006	SAP NetWeaver 7.1
2011	SAP NetWeaver 7.3
2013	SAP NetWeaver 7.4
2015	SAP NetWeaver 7.5

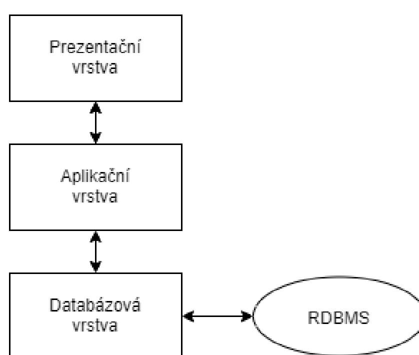
4.4 Architektura systému SAP

Pochopení architektury SAP systému z technického úhlu pohledu je klíčové pro pochopení běhu programů v jazyce ABAP. Systém SAP využívá třívrstvou architekturu typu klient-server.

První vrstvou je vrstva prezentační, která obstarává komunikaci s uživatelem ve formě uživatelského prostředí. Druhá vrstva, vrstva aplikační, má na starost obchodní logiku

celého systému. Poslední je databázová vrstva, která má na starost ukládání dat do databáze. Tyto vrstvy jsou navrženy a udržovány jako oddělené moduly, které se velmi často nachází na samostatných platformách (viz Obrázek 3).

Prezentační vrstva komunikuje s aplikační vrstvou a ta následně s databázovou vrstvou. Prezentační vrstva nekomunikuje s databázovou vrstvou napřímo, což ještě více zlepšuje bezpečnost obchodní logiky, jelikož klient nemá přímý přístup k databázi, respektive datům samotným. Oproti dvouvrstvé architektuře nabízí třívrstvá architektura možnost údržby a aktualizace každé vrstvy nezávisle na sobě. Ačkoliv je třívrstvá architektura složitější na implementaci, umožňuje vytvořit lépe a více škálovatelný systém.



Obrázek 3 - Schéma komunikace v systému SAP. Překresleno a přeloženo z [4].

Prezentační vrstva obsahuje uživatelské rozhraní, které běží na klientském zařízení. Uživatelským rozhraním může být například webový prohlížeč nebo speciální software, například SAP GUI. Data získána z uživatelského rozhraní jsou odeslána na aplikační server a získané výsledky jsou následně zobrazeny uživateli.

V aplikační vrstvě se o většinu operací starají programy v jazyce ABAP. Tato softwarová vrstva obsluhuje několik modulů obchodních funkcí a spouští programový kód. Aplikační vrstva může obsahovat jeden či více aplikačních serverů a server zpráv, jehož hlavní funkcí je komunikace v rámci vrstvy.

V databázové vrstvě jsou uložena všechna perzistentní data. Správu databáze většinou obstarává systém řízení relačních databází (RDBMS – Relational Database Management System). Samotná data jsou pak uložena v jedné či více tabulkách. Každý řádek tabulky je jednoznačně určen primárním klíčem. Vazby mezi tabulkami obstarávají cizí klíče. Systém SAP podporuje mnoho různých databázových systému, například SAP HANA, Oracle, Microsoft SQL a další.

4.5 Programovací jazyk ABAP

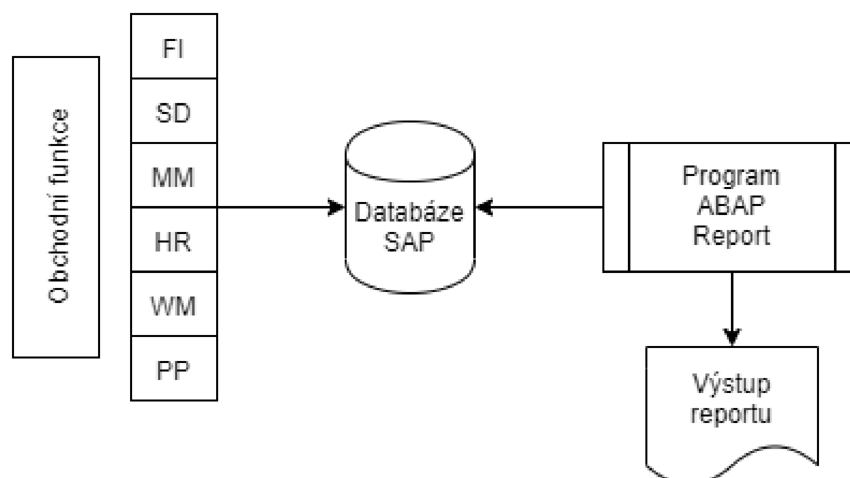
Programovací jazyk ABAP (Advanced Business Application Programming) je využíván pro vývoj aplikací v systému SAP. ABAP je hybridní programovací jazyk, který podporuje procedurální i objektově orientované programování. V současné době je možné vyvíjet aplikace v jazyce ABAP jak pro tradiční prostředí SAP GUI, tak i pro webové prostředí. Aplikace standardního SAP GUI jsou rozděleny na několik základních typů.

4.5.1 Základní typy aplikací

Základních typů aplikací je celkem 5. Reporty, rozhraní, konverzní programy, rozšíření a formuláře (akronym RICEF – Reports, Interfaces, Conversions, Extensions and Forms). Pro tuto práci jsou klíčové hlavně reporty, ostatní typy proto nejsou popsány.

Reporty

Reporty jsou určeny zejména k získávání a zobrazování dat z databáze. Data v SAP systému jsou aktualizována pomocí transakcí a dále zpracovávána do výstupních zpráv. V jazyce ABAP je rozlišováno celkem 10 typů programů: Pro tuto práci jsou zásadní pouze 3, a to spustitelné programy, programy typu include a tzv. module pool. Princip funkce reportů popisuje Obrázek 4



Obrázek 4 - Schéma principu funkce reportu. Překresleno a přeloženo z [4].

Spustitelné programy jsou určeny pro vývoj reportů kvůli pořadí provádění. Spustitelné programy začínají klíčovým slovem REPORT, které je následováno názvem programu. Spustitelný program může obsahovat libovolné procesní bloky jazyka ABAP. Mohou

také obsahovat libovolný počet lokálních tříd. Spustitelné programy mají podporu obrazovek, avšak jejich použití není pro běh programu nutné. Obrazovky je možné generovat pomocí specifických příkazů v kódu. Rozlišují se dva typy obrazovek. Výběrové obrazovky a obrazovky seznamu. Výběrové obrazovky přebírají parametry od uživatele, kdežto obrazovky seznamu zobrazují data na výstupu. Spustitelné programy jsou jediným typem programu, který může být použit pro zpracování dat na pozadí. Programy se spouští z transakce SA38, respektive SE38. Pro vytvoření a editaci programu slouží transakce SE80.

Programy typu include poskytují funkce knihoven, díky kterým je možné pomocí klíčového slova INCLUDE vložit zdrojový kód uchovávaný v programu typu include do jiného programu. Nejedná se o samostatné programy, tudíž je není možné nezávisle spouštět. Programy typu include slouží k organizaci zdrojového kódu hlavního programu do menších editovatelných jednotek.

Module pool je typ programu používaný pro dialogově zaměřené aplikace obsahující velké množství obrazovek. Modul pool lze vytvořit v transakci SE80. Obdobně jako spustitelné programy jsou uvozeny klíčovým slovem REPORT, je modul pool uveden klíčovým slovem PROGRAM. Modul pool může obsahovat všechny možné procesní bloky jazyku ABAP, kromě funkčních modulů a událostí reportů. Stejně jako spustitelné programy mohou obsahovat libovolný počet lokálních tříd. Na rozdíl od spustitelných programů není modul pool spouštěn pomocí názvu programu, ale je spouštěn pomocí kódu transakce. Pro každý modul pool musí být nadefinována minimálně jedna obrazovka. Tato obrazovka je zobrazena uživateli po zadání příslušného kódu transakce. Z hlavní obrazovky jsou poté spouštěny a volány jednotlivé moduly. Provedení modul poolu je řízeno logikou toku obrazovky. Obrazovky jsou vytvářeny pomocí speciálního nástroje Screen Painter. Popis programu typu module pool vychází z [7] a [8].

4.6 Verze jazyka ABAP

Aktuální verze jazyka ABAP, která byla vydána v září roku 2019, nese označení 7.54. [7] Zásadní verzí jazyka ABAP byla verze 7.4, vydaná v roce 2013. V této verzi došlo k radikálním změnám v zápisu syntaxe. Díky těmto změnám není možné spouštět programy zapsané ve verzi 7.4 a novější na systémech SAP, kde se nachází jazyk ABAP

ve verzi 7.3 a starší. Změny se týkaly zejména zjednodušení zápisu syntaxe, dále není nutné definovat datové typy proměnných a podobně. Všechny změny jsou podrobněji popsány v následující kapitole. Následující tabulka (Tabulka 2) stručně popisuje důležité změny v každé verzi. Uvedené informací vychází z [8].

Tabulka 2 - Vývoj verzí jazyka ABAP. Převzato a přeloženo z [13].

Verze	**Datum vydání**	**Důležité změny**
4.6C	5/2000	ABAP objekty
6.40	2004	Sdílené objekty
7.0	2006	Switch framework/Enhancement koncept
7.02		Výrazy typu: check lcl=>mi(1) + abs(-2) >= 3.
7.40	29/11/2012	"Code pushdown"—využití pokročilých funkcí ze základní databáze Operátory konstruktoru (NEW, VALUE, REF, CONV, CAST, EXACT, COND, SWITCH) Částečně implementované testy tříd Table Expressions Internal table line existence/line index functions
7.40 SP05	12/2013	MOVE-CORRESPONDING pro interní tabulky LET ve výrazech konstruktoru CORRESPONDING operátor CDS Views ABAP Managed Database Procedures
7.40 SP08	9/2014	výrazy FOR operátor konstruktoru REDUCE Výchozí hodnota pro výrazy konstruktoru GROUP BY pro interní tabulky Filtry a výchozí hodnoty pro tabulky Inline deklarace Open SQL po příkazu INTO

		Volitelné metody rozhraní
7.50	11/2015	Nový datový typ INT8 Relační výraz IS INSTANCE OF Hostitel výrazů Open SQL Open SQL výrazy ROUND, CONCAT, LPAD, LENGTH, REPLACE, RIGHT, RTRIM a SUBSTRING Výraz cesty Open SQL Open SQL SELECT UNION Open SQL INSERT FROM <poddotaz>
7.51	10/2016	Výčet Běžné tabulkové výrazy v Open SQL Open SQL/CDS CROSS JOIN Obsluha klienta CDS Rozšíření metadat v CDS Datum a čas v Open SQL/CDS Pohyblivá desetinná čárka v Open SQL/CDS
7.52	9/2017	Virtuální třídění tabulek
7.54	9/2019	Nové vestavěné datové typy Nový vestavěný datový typ utclong Přiřazení výpočtů Desetinná místa v časovém razítku Obsluha klienta Rozšíření klauzule INTO Definice asociací Nová agregační funkce STRING_AGG rozšíření DISTINCT volitelné v agregační funkci COUNT Window výrazy Revize podmínek SQL Dočasné hierarchie Nové hierarchické navigátory

		Agregační výrazy ve výrazech SQL Revize CAST matice Striktní režim při kontrole syntaxe Kontrola syntaxe literálů Vysvětlivky databázové nápovědy Vysvětlivky Releasing elementů Specifikace CDS entit za USING Uvozovky pro logická schémata Rozšířen ... unique Externí pojmenování akcí Kontrola statických polí Jazyk pro manipulaci s entitami TYPE STRUCTURE FOR Informace o odpovědi Absolutní názvy typů Business služba Kontrola autorizace během aktualizace Implicitní typ zpráv v IF_T100_DYN_MSG Užití testovacích tříd Rozšíření COMBINATION MODE OR|AND příkazu GRANT SELECT ON Rozšíření REDEFINITION příkazu GRANT SELECT ON predikát VOID Rozšíření IN SCENARIO příkazu GRANT SELECT ON Podmínka obecného aspektu příkazu DEFINE ROLE Nová varianta INHERITING CONDITIONS FROM SUPER Odlišné operátory REPLACING Definice obecného aspektu DCL omezení pro hierarchie ABAP CDS DCL omezení pro entity transakčního projekčního zobrazení

4.6.1 Změny ve verzi 7.4 a porovnání se starší verzí

V této podkapitole jsou popsány veškeré změny týkající se jazyka ABAP s příchodem verze 7.4. Jednotlivé změny jsou pro lepší přehlednost rozděleny do dílčích skupin. Každá skupina obsahuje popis změny a příklady syntaxe ve verzi jazyka do 7.3 a od 7.4. Popis změn a porovnání se starší verzí vychází z [7] a [8].

Inline deklarace

Inline deklarace proměnných umožňuje definovat proměnnou až v okamžiku jejího použití. Například ve smyčkách, přiřazení hodnoty a podobně. Dále není nutné explicitně definovat datový typ proměnné. Datový typ je určen automaticky podle hodnoty proměnné a práce s ní během vykonávání programu. V následující tabulce se nachází příklady inline deklarací.

Tabulka 3 - Tabulka změn - Inline deklarace. Převzato z [8].

Popis	7.3 a starší	7.4 a novější
Příkaz DATA – deklarace proměnné a přiřazení hodnoty	DATA text TYPE string. Text = 'ABC'	DATA(text) = 'ABC'.
Smyčka LOOP	DATA wa LIKE LINE OF itab. LOOP AT itab INTO wa. … ENDLOOP.	LOOP AT itab INTO DATA(wa). … ENDLOOP.
Volání metod	DATA a1 TYPE … DATA a2 TYPE … oref->meth(IMPORTING p1 = a1 IMPORTING p2 = a2).	oref->meth(IMPORTING p1 = DATA(a1) IMPORTING p2 = DATA(a2)).

Smyčka LOOP s přiřazením	FIELD-SYMBOLS: <line> TYPE … LOOP AT itab ASSIGNING <line>. … ENDLOOP.	LOOP AT itab ASSIGNING FIELD-SYMBOL(<line>). … ENDLOOP.
Čtení interní tabulky s přiřazením	FIELD-SYMBOLS: <line> TYPE … READ TABLE itab ASSIGNING <line>.	READ TABLE itab ASSIGNING FIELD-SYMBOL(<line>).
Čtení databázové tabulky – příkaz SELECT	DATA itab TYPE TABLE OF dbtab. SELECT * FROM dbtab INTO TABLE itab WHERE fld1 = lv_fld1.	SELECT * FROM dbtab INTO TABLE DATA(itab) WHERE fld1 = @lv_fld1.
Čtení jednoho řádku databázové tabulky – příkaz SELECT SINGLE	SELECT SINGLE f1 f2 FROM dbtab INTO (lv_f1, lv_f2) WHERE … WRITE: / lv_f1, lv_f2.	SELECT SINGLE f1 AS my_f1, f2 AS abc FROM dbtab INTO DATA(ls_structure) WHERE … WRITE: / ls_structure-my_f1, ls_structure-abc.

Práce s tabulkami

Práce s daty v tabulkách, respektive s tabulkami obecně, byla s příchodem verze 7.4 značně zjednodušena. Novinkou je výjimka CX_SY_ITAB_LINE_NOT_FOUND, která je vyvolána, pokud požadovaný řádek není v tabulce nalezen. Ve starších verzích jazyka bylo nutné kontrolovat stav systémové proměnné sy-subrc.

Tabulka 4 - Tabulka změn - Práce s tabulkami. Převzato z [8].

Popis	7.3 a starší	7.4 a novější
Čtení konkrétního indexu tabulky	READ TABLE itab INDEX idx INTO wa.	wa = itab[idx].

Čtení z tabulky pomocí klíče	READ TABLE itab INDEX idx USING KEY key INTO wa.	wa = itab[KEY key INDEX idx].
Čtení tabulky s klíčem	READ TABLE itab WITH KEY col1 = … col2 = … INTO wa.	wa = itab[col1 = … col2 = …].
Čtení tabulky s klíčovými komponenty	READ TABLE itab WITH TABLE KEY key COMPONENTS col1 = … col2 = … INTO wa.	wa = itab[KEY key col1 = … col2 = …].
Existence řádku	READ TABLE itab … TRANSPORTING NO FIELDS. IF sy-subrc = 0. … ENDIF.	IF line_exists(itab[…]). … ENDIF.
Získání indexu tabulky	DATA idx TYPE sy-tabix. READ TABLE … TRANSPORTING NO FIELDS. idx = sy-tabix.	DATA(idx) = line_index(itab[…]).

Operátor CONV

Nově přidaný operátor CONV slouží ke konverzi mezi datovými typy. Konverzi proměnné lze zapsat dvěma způsoby. Lze použít konkrétní datový typ podporovaný jazykem ABAP, nebo je možné ponechat rozhodnutí o výběru datového typu na kompilátoru.

CONV dtype|# (…), kde dtype představuje konkrétní datový typ. Pokud je výhodnější ponechat rozhodnutí o datovém typu na kompilátoru, místo identifikátoru dtype je použit znak #. Následuje název proměnné v kulatých závorkách.

Tabulka 5 - Tabulka změn - CONV. Převzato z [8].

7.3 a starší

`DATA text   TYPE c LENGTH 255.`
`DATA helper TYPE string.`
`DATA xstr   TYPE xstring.`
 	
`helper = text.`
`xstr = cl_abap_codepage=>convert_to( source = helper ).`

7.4 a novější

`DATA text TYPE c LENGTH 255.`
 	
`DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV string( text ) ).`
OR
`DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV #( text ) ).`

Operátor VALUE

Operátor VALUE je používán pro vytvoření počáteční hodnoty libovolného neobecného datového typu. Dále slouží ke konstrukci obsahu strukturovaných datových typů a tabulek.

```
TYPES:  BEGIN OF ty_columns1, "Simple structure
           cols1 TYPE i,
           cols2 TYPE i,
        END OF ty_columns1.
TYPES:  BEGIN OF ty_columnns2,  "Nested structure
           coln1 TYPE i,
           coln2 TYPE ty_columns1,
        END OF ty_columns2.
DATA: struc_simple TYPE ty_columns1,
      struc_nest   TYPE ty_columns2.
struct_nest    = VALUE t_struct(coln1 = 1
                               coln2-cols1 = 1
                               coln2-cols2 = 2 ).
```

NEBO

```
struct_nest    = VALUE t_struct(coln1 = 1
                               coln2 = VALUE #( cols1 = 1
                                                cols2 = 2 ) ).
```

Zdrojový kód 1 - Přehled změn - VALUE. Převzato z [8]

Operátor FOR

Operátor FOR představuje takzvaný iterační výraz, který je používán jako podvýraz výrazů konstruktoru. Spolu s redukčním operátorem REDUCE, instančním operátorem NEW a operátorem VALUE slouží pro vytváření interních tabulek.

Tabulka 6 - Tabulka změn - FOR. Převzato z [8].

7.3 a starší

`DATA: gt_citys TYPE ty_citys,`
`       gs_ship  TYPE ty_ship,`
`       gs_city  TYPE ort01.`
`LOOP AT gt_ships INTO gs_ship.`
`  gs_city =  gs_ship-city.`
`  APPEND gs_city TO gt_citys.`
`ENDLOOP.`

7.4 a novější
`DATA(gt_citys) = VALUE ty_citys( FOR ls_ship IN gt_ships ( ls_ship-city ) ).`

Operátor FOR lze doplnit operátory UNTIL nebo WHILE, které umožňují podmíněné iterace. Iterační podmínka je logický výraz. V mnoha případech mohou podmíněné iterace nahradit smyčky DO-WHILE.

Tabulka 7 - Tabulka změn - DO-WHILE. Převzato z [8].

<table>
<tr><th>7.3 a starší</th></tr>
<tr><td><pre>DATA: gt_itab TYPE ty_tab,
      j   TYPE i.
FIELD-SYMBOLS <ls_tab> TYPE ty_line.
j = 1.
DO.
j = j + 10.
IF j > 40. EXIT. ENDIF.
APPEND INITIAL LINE TO gt_itab ASSIGNING <ls_tab>.
<ls_tab>-col1 = j.
<ls_tab>-col2 = j + 1.
<ls_tab>-col3 = j + 2.
ENDDO.</pre></td></tr>
<tr><th>7.4 a novější</th></tr>
<tr><td><pre>DATA(gt_itab) = VALUE ty_tab(FOR j = 11 THEN j + 10 UNTIL
j > 40   (col1 = j col2 = j + 1 col3 = j + 2)).</pre></td></tr>
</table>

Operátor REDUCE

Redukční operátor REDUCE je obecně používán pro redukci datové množiny iteračních výrazů na jeden datový prvek. V případě užití operátoru REDUCE nad tabulkovou iterací, je výsledkem redukce souhrnná hodnota, například počet řádků splňujících zadanou podmínku.

Tabulka 8 - Tabulka změn – REDUCE 1. Převzato z [8].

<table>
<tr><th>7.3 a starší</th></tr>
<tr><td><pre>DATA: lv_lines TYPE i.
LOOP AT gt_itab INTO ls_itab WHERE F1 = ‘XYZ’.
  lv_lines = lv_lines + 1.</pre></td></tr>
</table>

ENDLOOP.
7.4 a novější
`DATA(lv_lines) = REDUCE i( INIT x = 0 FOR wa IN gt_itab WHERE( F1 = 'XYZ' ) NEXT x = x + 1 ).`

Tabulka 9 - Tabulka změn - REDUCE 2. Převzato z [8].

7.3 a starší

`DATA: lv_line TYPE i,`
`      lv_sum  TYPE i.`
 	
`LOOP AT gt_itab INTO lv_line.`
`  lv_sum = lv_sum + lv_line.`
`ENDLOOP.`

7.4 a novější
`DATA(lv_sum) = REDUCE i( INIT x = 0 FOR wa IN itab NEXT x = x + wa ).`

Operátory COND a SWITCH

Operátory COND a SWITCH spadají do takzvaných podmíněných výrazů. Podmíněné výrazy vytváří a přiřazují hodnotu, nebo vyvolávají výjimky tříd. Chování je specifikováno logickými výrazy nebo případovým rozlišením. Na první pohled je využití operátoru COND a SWITCH stejné. Hlavní rozdíl je v tom, že operátor COND využívá v jednotlivých větvích WHEN logických výrazů, kdežto SWITCH využívá konstanty.

```
DATA(time) =
     COND string( WHEN sy-timlo < '120000' THEN
                      |{ sy-timlo TIME = ISO } AM|
                  WHEN sy-timlo > '120000' THEN
                      |{ CONV t( sy-timlo - 12 * 3600 )TIME = ISO } PM|
                  WHEN sy-timlo = '120000' THEN
                      |High Noon|
                  ELSE
                      THROW cx_cant_be( ) ).
```

Zdrojový kód 2 - Přehled změn - COND. Převzato z[8].

```
DATA(text) =
     NEW class( )->meth( SWITCH #( sy-langu
                              WHEN 'D' THEN `DE`
                              WHEN 'E' THEN `EN`
                              ELSE THROW cx_langu_not_supported( ) ) ).
```

Zdrojový kód 3 - Přehled změn - SWITCH. Převzato z [8].

Operátor CORRESPONDING

Operátor CORRESPONDING slouží pro sloučení datových struktur. Přesunou se pouze ty položky/sloupce, které existují v obou datových strukturách.

Tabulka 10 - Tabulka změn - CORRESPODING. Převzato z [8].

	7.3 a starší	7.4 a novější

1	`CLEAR ls_line2.`
	
	`MOVE-CORRESPONDING ls_line1 TO ls_line2.`
		`ls_line2 = CORRESPONDING #( ls_line1 ).`

2	`MOVE-CORRESPONDING ls_line1 TO ls_line2`	`ls_line2 =` `CORRESPONDING #BASE ( ls_line2 ) ls_line1 ).`

3	`DATA: ls_line3 LIKE ls_line2.`
	
	`ls_line3 = ls_line2.`
	
	`MOVE-CORRESPONDING ls_line1 TO ls_line2.`
		`DATA(ls_line3) = CORRESPONDING`
		`line2 BASE ( ls_line2 ) ls_line1 ).`

Textové řetězce

Jednou z novinek jsou textové šablony. Textová šablona je ohraničena znakem |. Mezi těmito symboly se může nacházet libovolný text. Mezi složenými závorkami {} se mohou nacházet například datové objekty, výpočty, výrazy konstruktoru atd. Díky šablonám je nyní snazší i spojování řetězců. V rámci textových šablon lze nastavit šířku, zarovnání a okraje při výpisu na obrazovku pomocí příkazu WRITE. Ovlivnit je možné také velikost

znaků (všechna velká, všechna malá). V textových šablonách lze pomocí klíčového slova ALPHA ovlivnit zobrazení číslic na výstupu či pomocí DATE měnit formát data.

```
WRITE / |{ 'Left'     WIDTH = 20 ALIGN = LEFT   PAD = '0' }|.
WRITE / |{ 'Centre'   WIDTH = 20 ALIGN = CENTER PAD = '0' }|.
WRITE / |{ 'Right'    WIDTH = 20 ALIGN = RIGHT  PAD = '0' }|.


WRITE / |{ 'Text' CASE = (cl_abap_format=>c_raw) }|.
WRITE / |{ 'Text' CASE = (cl_abap_format=>c_upper) }|.
WRITE / |{ 'Text' CASE = (cl_abap_format=>c_lower) }|.


DATA(lv_vbeln) = '0000012345'.
WRITE / |{ lv_vbeln  ALPHA = OUT }|.   "or use ALPHA = IN to go in
other direction


WRITE / |{ pa_date DATE = ISO }|.           "Date Format YYYY-MM-DD
WRITE / |{ pa_date DATE = User }|.          "As per user settings
WRITE / |{ pa_date DATE = Environment }|.   "Formatting setting of
language environment
```

Zdrojový kód 4 - Přehled změn - Textové řetězce. Převzato z [8].

Tabulka 11 - Tabulka změn - Textové řetězce 1. Převzato z [8].

<table>
<tr><th>7.3 a starší</th></tr>
<tr><td><pre>DATA itab TYPE TABLE OF scarr.

SELECT * FROM scarr INTO TABLE itab.

DATA wa LIKE LINE OF itab.

READ TABLE itab WITH KEY carrid = 'LH' INTO wa.

DATA output TYPE string.

CONCATENATE 'Carrier:' wa-carrname INTO output SEPARATED BY SPACE.

cl_demo_output=>display(output).</pre></td></tr>
<tr><th>7.4 a novější</th></tr>
<tr><td><pre>SELECT * FROM scarr INTO TABLE @DATA(lt_scarr).
cl_demo_output=>display(|Carrier: { lt_scarr[carrid = 'LH']-carrname }|).</pre></td></tr>
</table>

Tabulka 12 - tabulka změn - Textové řetězce 1. Převzato z [8]

<table>
<tr><th>7.3 a starší</th></tr>
<tr><td><pre>DATA lv_output TYPE string.
CONCATENATE 'Hello' 'world' INTO lv_output SEPARATED BY SPACE.</pre></td></tr>
<tr><th>7.4 a novější</th></tr>
<tr><td><pre>DATA(lv_out) = |Hello| & | | & |world|.</pre></td></tr>
</table>

GROUP BY ve smyčkách LOOP

Klauzule GROUP BY rozdělí tabulku na skupiny odpovídající klíči. Počet iterací smyčky LOOP odpovídá počtu skupin v tabulce. Nad každou skupinou lze poté iterovat pomocí příkazu LOOP AT GROUP. Objekt skupiny kromě jednotlivých položek obsahuje také informaci o počtu položek ve skupině a index skupiny.

```
TYPES: BEGIN OF ty_employee,
  name TYPE char30,
  role     TYPE char30,
  age     TYPE i,
END OF ty_employee,

ty_employee_t TYPE STANDARD TABLE OF ty_employee WITH KEY name.

DATA(gt_employee) = VALUE ty_employee_t(
( name = 'John'      role = 'ABAP guru'        age = 34 )
( name = 'Alice'      role = 'FI Consultant'   age = 42 )
( name = 'Barry'     role = 'ABAP guru'        age = 54 )
( name = 'Mary'      role = 'FI Consultant'   age = 37 )
( name = 'Arthur'    role = 'ABAP guru'        age = 34 )
( name = 'Mandy'  role = 'SD Consultant'  age = 64 ) ).

DATA: gv_tot_age TYPE i, gv_avg_age TYPE decfloat34.

"Loop with grouping on Role
LOOP AT gt_employee INTO DATA(ls_employee)
GROUP BY ( role   = ls_employee-role
           size   = GROUP SIZE
           index = GROUP INDEX )
ASCENDING ASSIGNING FIELD-SYMBOL(<group>).

  CLEAR: gv_tot_age.
  "Output info at group level
  WRITE: / |Group: { <group>-index }
            Role: { <group>-role WIDTH = 15 }|
         & |Number in this role: { <group>-size }|.

   "Loop at members of the group
   LOOP AT GROUP <group> ASSIGNING FIELD-SYMBOL(<ls_member>).
      gv_tot_age = gv_tot_age + <ls_member>-age.
      WRITE: /13 <ls_member>-name.
   ENDLOOP.

   "Average age
   gv_avg_age = gv_tot_age / <group>-size.
   WRITE: / |Average age: { gv_avg_age }|.
   SKIP.
ENDLOOP.
```

Zdrojový kód 5 - Přehled změn - GROOP & LOOP 1. Převzato z [8].

```
Group: 1    Role: ABAP guru          Number in this role: 3
            John
            Barry
            Arthur
Average age: 40.666666666666666666666666666666667
Group: 2    Role: FI Consultant      Number in this role: 2
            Alice
            Mary
Average age: 39.5
Group: 3    Role: SD Consultant      Number in this role: 1
            Mandy
Average age: 64
```

Zdrojový kód 6 - Přehled změn - GROUP BY & LOOP 2. Převzato z [8].

Třídy a metody

První změnou je přímé odkazování polí v rámci vrácených datových struktur. To vede k zjednodušení syntaxe. Dále odpadá nutnost porovnávat výsledek metody s návratovou hodnotou typu BOOLEAN. Poslední novinkou je využití operátoru NEW pro vytvoření nové instance objektu.

Tabulka 13 - Tabulka změn - Třídy a metody 1. Převzato z [8].

7.3 a starší

`DATA: ls_lfa1 TYPE lfa1, lv_name1 TYPE lfa1-name1.`
 	
`ls_lfa1  = My_Class=>get_lfa1( ).`
`lv_name1 = ls_lfa1-name1.`

7.4 a novější
`DATA(lv_name1) = My_Class=>get_lfa1( )-name1.`

Tabulka 14 - Tabulka změn - Třídy a metody 2. Převzato z [8].

7.3 a starší

`IF My_Class=>return_boolean( ) = abap_true.`
…
`ENDIF.`

7.4 a novější

```
IF My_Class=>return_boolean( ).
…
ENDIF.
```

Tabulka 15 - Tabulka změn - Třídy a metody 3. Převzato z [8].

7.3 a starší

`IF My_Class=>return_boolean( ) = abap_true.`
…
`ENDIF.`

7.4 a novější

`IF My_Class=>return_boolean( ).`
…
`ENDIF.`

Tabulka 16 - Tabulka změn - Třídy a metody 4. Převzato z [8].

7.3 a starší

`DATA: lo_delivs TYPE REF TO zcl_sd_delivs,`
`              lo_deliv  TYPE REF TO zcl_sd_deliv.`
 	
`CREATE OBJECT lo_delivs.`
`CREATE OBJECT lo_deliv.`
 	
`lo_deliv = lo_delivs->get_deliv( lv_vbeln ).`

7.4 a novější
`DATA(lo_deliv) = new zcl_sd_delivs( )->get_deliv( lv_vbeln ).`

Datová struktura Mesh

Mesh je datová struktura umožňující vytvoření asociace mezi příbuznými datovými skupinami.

Příklad - máme dvě tabulky: tabulku manažerů a tabulku zaměstnanců, přičemž v tabulce zaměstnanců je ke každému zaměstnanci přiřazen manažer. Chceme-li vypsat informace o manažerovi některého zaměstnance, nebo pro některého manažera vypsat všechny jeho zaměstnance, spojíme obě tabulky pomocí meshe.

```
TYPES: BEGIN OF MESH m_team,
    managers TYPE tt_manager  ASSOCIATION my_employee TO developers
             ON manager = name,
    developers TYPE tt_developer ASSOCIATION my_manager TO managers
               ON name = manager,
END OF MESH m_team.

DATA: ls_team TYPE m_team.
ls_team-managers   = lt_manager.
ls_team-developers = lt_developer.

*Get details of Jerry's manager *
"get line of dev table
ASSIGN lt_developer[ name = 'Jerry' ] TO FIELD-SYMBOL(<ls_jerry>).
DATA(ls_jmanager) =  ls_team-developers\my_manager[ <ls_jerry> ].
WRITE: / |Jerry's manager: { ls_jmanager-name }|,30
         |Salary: { ls_jmanager-salary }|.

"Get Thomas' developers
SKIP.
WRITE: / |Thomas' developers:|.

"line of manager table
ASSIGN lt_manager[ name = 'Thomas' ] TO FIELD-SYMBOL(<ls_thomas>).
LOOP AT ls_team-managers\my_employee[ <ls_thomas> ]
  ASSIGNING FIELD-SYMBOL(<ls_emp>).

  WRITE: / |Employee name: { <ls_emp>-name }|.
ENDLOOP.
```

Zdrojový kód 7 - Přehled změn - Mesh 1. Převzato z [8].

```
Jerry's manager: Jason     Salary: 3000
Thomas' developers:
    Employee name: David
    Employee name: Jack
    Employee name: John
```

Zdrojový kód 8 - Přehled změn - Mesh 2. Převzato z [8].

Filtry

Filtry slouží k filtrování záznamů tabulky na základě záznamu v jiné tabulce. Tabulky sloužící jako filtry jsou označovány jako filtrovací tabulky. Pro zobrazení záznamů, které vyhovují filtrovací tabulce je použito klíčové slovo IN, opačné záznamy je možné získat pomocí klíčového slova EXCEPT.

```
TYPES: BEGIN OF ty_filter,
         cityfrom TYPE spfli-cityfrom,
         cityto   TYPE spfli-cityto,
         f3       TYPE i,
       END OF ty_filter,
       ty_filter_tab TYPE HASHED TABLE OF ty_filter
                     WITH UNIQUE KEY cityfrom cityto.

DATA: lt_splfi TYPE STANDARD TABLE OF spfli.
SELECT * FROM spfli APPENDING TABLE lt_splfi.

DATA(lt_filter) = VALUE ty_filter_tab( f3 = 2
                  ( cityfrom = 'NEW YORK'  cityto  = 'SAN FRANCISCO' )
                  ( cityfrom = 'FRANKFURT' cityto  = 'NEW YORK' )   ).

DATA(lt_myrecs) = FILTER #( lt_splfi IN lt_filter
                            WHERE cityfrom = cityfrom
AND cityto = cityto ).

"Output filtered records
LOOP AT lt_myrecs ASSIGNING FIELD-SYMBOL(<ls_rec>).
  WRITE: / <ls_rec>-carrid,8 <ls_rec>-cityfrom,30
           <ls_rec>-cityto,45 <ls_rec>-deptime.
ENDLOOP.
```

Zdrojový kód 9 - Přehled změn - Filtry. Převzato z [8].

5 Gramatika jazyka ABAP

Gramatika je nedílnou součástí překladače. Jelikož výrobce systému SAP neposkytuje gramatiku jazyka ABAP ve veřejných zdrojích, bylo nutné ji celou vytvořit. Přestože byly změny v syntaxi natolik rozsáhlé, že došlo ke ztrátě zpětné kompatibility, je většina pravidel shodná pro obě verze jazyka. Respektive jejich množiny, verze 7.3 a starší, a 7.4 a novější.

Při tvorbě gramatik jsem vycházel z dokumentace pro jazyk ABAP [7]. V dokumentaci jsou vypsána všechna klíčová slova, kterých je kolem tisíce. Tato klíčová slova jsou v gramatice použita jako znaky abecedy. Dokumentace dále obsahuje předpis pro tvorbu názvů proměnných a funkcí a povolené matematické operátory. Gramatiky pro oba jazyky jsou zapsané ve formátu EBNF.

5.1 Pravidla gramatiky a jejich popis

Ke každému pravidlu je uveden jeho popis, včetně vysvětlení jednotlivých terminálních či neterminálních symbolů. Dále je uveden předpis pravidla v normě EBNF.

5.1.1 Počáteční pravidlo gramatiky

Počáteční pravidlo, vstupní bod překladu obsahuje libovolný počet neterminálních symbolů *statement*, které představují jednotlivé příkazy jazyka ABAP.

```
prog : statement*;
```

5.1.2 Příkaz jazyka ABAP

Pravidlo pro příkazy jazyka ABAP se skládá z neterminálních symbolů představujících již konkrétní typy příkazů. Hlavním důvodem této formulace je přehlednost v gramatice. Specifické příkazy jako například IF, LOOP a další jsou popsány v následujících kapitolách.

```
statement :
  ( generic | declaration | assignment | loop_into | loop_assign |
  method_call | read_assign | select_into | select_into | st_if |
  st_case | comment | table_index );
```

Příkazy, které nevyžadují speciální kontrolu zápisu lze označit za obecné. Tyto příkazy se skládají z následujících neterminálních symbolů: *keyword*, *operator*, *operand*, *addition*. *Keyword* představuje některé z množiny klíčových slov jazyka ABAP. Jak již název napovídá, *operator* ukrývá některý z operátorů jazyka ABAP. *Operand* je neterminální symbol představující uživatelsky definovanou proměnnou. V poslední řadě *addition* je rozšířením *keyword*, tedy klíčového slova jazyka ABAP. Každé klíčové slovo má svou specifickou množinu rozšiřujících klíčových slov. *Keyword* a *addition* představují množinu terminálních symbolů, klíčových slov jazyka ABAP. Klíčové slovo se v příkazu vyskytuje právě jednou. Následuje libovolná kombinace operátorů, operandů či rozšířeních klíčového slova. Každý příkaz je zakončen tečkou, respektive terminálním symbolem *DOT*.

```
generic : ( operator | operand | addition )* DOT;
```

5.1.3 Operandy

Operandy neboli uživatelsky definované proměnné se skládají z terminálních symbolů. Zejména tedy znaků abecedy, číslic, podtržítek nebo pomlček. Tyto znaky jsou v gramatice označovány jako terminální symboly *CHAR_STR* a *INTEGER*. Operandy mohou mít takzvaný prefix, který je rovněž tvořen výše zmíněnými terminálními symboly. Název prefixu je ohraničen lomítky a jeho minimální délka činí 3 znaky. Operandy i názvy prefixů mohou začínat pouze velkým nebo malým písmenem.

Zápis pravidla pro operandy, respektive uživatelské proměnné, má tento zápis, protože jeho název může obsahovat libovolné z klíčových slov či datových typů. Samozřejmě musí být doplněno o libovolný další znak zleva či zprava, aby nedocházelo ke konfliktům při překladu.

```
operand : user_defined;
```

```
user_defined :
  (PREFIX)? ( CHAR_STR+ (CHAR_STR | INTEGER)* |
  CHAR_STR+ INTEGER* (data_type | ABAP_WORD) |
  (data_type | ABAP_WORD) (CHAR_STR | INTEGER)+ |
  CHAR_STR+ INTEGER* (data_type | ABAP_WORD) (CHAR_STR | INTEGER)+ |
  CHAR_STR* (data_type | ABAP_WORD) (CHAR_STR | INTEGER)* (data_type
  | ABAP_WORD)+ (CHAR_STR | INTEGER)* |
  CHAR_STR+ INTEGER* (data_type | ABAP_WORD) (CHAR_STR | INTEGER)*
  (data_type | ABAP_WORD)+ (CHAR_STR | INTEGER)* );


PREFIX : SLASH CHAR_STR CHAR_STR CHAR_STR+ SLASH
```

5.1.4 Operátory

Operátory lze v jazyce ABAP rozdělit do několika kategorií. Mezi tyto kategorie patří například deklarační operátory, aritmetické operátory, relační operátory a podobně. Pravidlo pro operátory se proto skládá z terminálních symbolů zastupujících právě tyto kategorie. To hlavně z důvodu přehlednosti gramatiky.

```
operator:
  ( DECLARATION_OP | CONSTRUCTOR_OP | ASSIGNMENT_OP | ARITHMETIC_OP
  | BIT_OP | STRING_OP | RELATIONAL_OP | BOOLEAN_OP | LITERAL_OP );
```

Deklarační operátory

Deklarační operátory jsou využívány k deklaraci proměnných. V jazyce ABAP jsou deklarační operátory dva. *DATA* a *FIELD-SYMBOL*, přičemž první zmíněný je určen k deklaraci proměnných. *FIELD-SYMBOL* je obdobou pointerů v jazyce C, odkazují na konkrétní místo v paměti.

```
DECLARATION_OP: DATA | FIELDSYMBOL;
```

Operátory konstruktoru

Operátory konstruktoru se používají při konstrukci nových hodnot, které mohou být dále předávány (*NEW*, *VALUE*) nebo mohou sloužit ke konverzi předané hodnoty (*CONV*, *CAST*, *REF*, *EXACT*). Operátory *NEW*, *CAST* a *REF* vždy vrací referenční proměnné, kdežto operátory VALUE, CONV a EXACT vrací různé typy datových objektů.

```
CONSTRUCTOR_OP :
  NEW | VALUE | CONV | CORRESPONDING | CAST | REF | EXACT | REDUCE |
  FILTER | COND | SWITCH;
```

Operátory přiřazení

Používanými operátory přiřazení v jazyce ABAP jsou = a *?=*. Operátor = se používá k prostému přiřazení hodnoty k proměnné. Operátor *?=* slouží k přetypování proměnných.

```
ASSIGNMENT_OP : EQ_OP | '?=';
```

Aritmetické operátory

Mezi aritmetické operátory v jazyce ABAP patří operátory pro klasické matematické operace, jako je sčítání, odčítání, násobení a dělení. Dalšími operátory jsou *DIV*, *MOD* a **. Operátor *DIV* vrací zaokrouhlený výsledek po dělení. Operátor *MOD* vrací zbytek po celočíselném dělení. Operátor ** se používá pro umocňování, tedy 5**3 je rovno 125.

```
ARITHMETIC_OP :
  PLUS_OP | MINUS_OP | MULTIPLY_OP | DIVISION_OP | DIV | MOD | '**';
```

Operátory bitových operací

Bitové operátory jsou použity pro provádění bitových logických operací jako bitový logický součin, součet, negace a non-ekvivalence.

```
BIT_OP : BITAND | BITOR | BITXOR | BITNOT;
```

Operátory pro práci s řetězci

Jediným operátorem určeným pro práci s řetězci je operátor &&. Tento operátor slouží k řetězení dvou a více řetězců. Jedná se v podstatě o obdobu metody *CONCATENATE*, kde je jako oddělovač použit prázdný znak. Pokud budeme mít řetězec A obsahující slovo „Hello“ a řetězec B obsahující slovo „World“, výsledkem operace A && B bude řetězec „HelloWorld“.

```
STRING_OP : '&&';
```

Relační operátory

Relační operátory se vyskytují v relačních výrazech a tvoří logické výrazy. Logické výrazy se vyskytují v podmínkách pro řízení programového toku a v podmínkách *WHERE* při zpracovávání interních tabulek. Tyto operátory spojují a mezi sebou porovnávají dva nebo více operandů libovolného datového typu. Jelikož jazyk ABAP

nepodporuje žádné logické datové typy, nelze výsledek logických operací přiřadit jako hodnotu proměnné.

```
RELATIONAL_OP : EQ_OP | '<>' | LT_OP | GT_OP | '<=' | '>=' |
BETWEEN;
```

Logické operátory

Logické operátory jsou použity pro provádění logických operací jako logický součin, součet, negace a non-ekvivalence.

```
BOOLEAN_OP : AND | OR | XOR | NOT;
```

Operátory literálů

Operátor literálů & je používán ke spojení dvou literálů nebo řetězcových šablon s literálem nebo řetězcovou šablonou. Pomocí operátoru *&* lze spojovat pouze literály do délky 255 znaků. Delší řetězce lze za běhu spojit pouze pomocí operátoru řetězení *&&*.

```
LITERAL_OP : '&';
```

5.1.5 Datové typy

Stejně jako operátory lze i datové typy rozdělit do několika kategorií. Jedná se například o číselné datové typy, znakové datové typy, datové typy pro uchování data a času a podobně. Pravidlo pro datové typy se, stejně jako u pravidla operátorů, skládá z terminálních symbolů zastupující jednotlivé kategorie datových typů.

```
DATA_TYPE :
  ( NUMERIC_TYPE | CHARACTER_TYPE | BYTE_TYPE | DATETIME_TYPE );


NUMERIC_TYPE :
  ( 'b' | 's' | 'i' | 'int8' | 'p' | 'decfloat16' | 'decfloat34' |
  'f' );

CHARACTER_TYPE :
  ( 'c' (WS LENGTH WS INTEGER)? | 'n' (WS LENGTH WS INTEGER)? |
  'string' );

BYTE_TYPE : ( 'x' | 'xstring' );

DATETIME_TYPE : ( ' d' | ' t' );
```

5.1.6 Deklarace proměnných

Od verze 7.40 lze v jazyce ABAP deklarovat proměnné dvěma způsoby. První, převzatý ze starší verze jazyka vyžaduje uvést při deklaraci konkrétní datový typ. Deklarace proměnné tímto způsobem začíná jedním z deklaračních operátorů. Následuje neterminální symbol *user_defined* označující název proměnné. Dále pak klíčové slovo *TYPE* doplněné o jeden z výše vypsaných datových typů. Druhý způsob deklarace, který byl zaveden ve verzi 7.40, konkrétní datový typ nevyžaduje. Užití jednoho z deklaračních operátorů zůstává. V tomhle případě je však následován pouze neterminálním symbolem *user_defined* v kulatých závorkách.

```
declaration :
  DECLARATION_OP user_defined TYPE data_type DOT |
  DECLARATION_OP LPAREN user_defined RPAREN DOT;
```

5.1.7 Přiřazení hodnoty proměnné

V jazyce ABAP lze přiřazení hodnoty proměnné rozdělit na dva typy. Prvním typem je přiřazení konkrétní hodnoty, obsahu jiné proměnné či matematického výrazu. Ve druhém případě se jedná o přiřazení hodnoty z interní tabulky na konkrétním indexu. V obou případech přiřazení začíná neterminálním symbolem *user_defined* následovaným znakem rovnosti. Jedná-li se o klasické přiřazení následuje neterminální symbol *value*, který označuje konkrétní hodnotu, nebo název jiné proměnné zastoupený neterminálním symbolem *user_defined*. V případě přiřazení hodnoty z tabulky následuje za znakem rovnosti neterminální symbol *user_defined* symbolizující název tabulky a v hranatých závorkách index. Ten je dle typu buď zastoupen neterminálním symbolem *user_defined* nebo celočíselnou hodnotou.

```
assignment : user_defined ASSIGNMENT_OP ( value | user_defined) DOT;

table_index :
  user_defined ASSIGNMENT_OP user_defined SQLPAREN (user_defined |
  INTEGER) SQRPAREN DOT;
```

5.1.8 Hodnota proměnné

V případě přiřazení konkrétní hodnoty proměnné lze rozlišit tři základní typy hodnot. Přiřadit lze číselnou hodnotu, ať už celočíselnou nebo s desetinnou čárkou, textový řetězec, nebo matematický výraz.

```
value: ( numeric | STRING | math_expression );

numeric: ( INTEGER | FLOAT );

INTEGER: NUM | NUM1 NUM*;

FLOAT: INTEGER DOT NUM+;

STRING:
  '\'' ( 'a'..'z' | 'A'..'Z' | '0'..'9'  | '_'  | ' ' |
  '.'  | ','  | ':' | '-')* '\'';

math_expression :
  t ARITHMETIC_OP math_expression | t;

t : f ARITHMETIC_OP t | f;

f: LPAREN math_expression RPAREN | numeric;
```

5.1.9 Smyčka LOOP

Smyčka LOOP v jazyce ABAP je obdobou cyklu *for each* používaného v jiných jazycích. Existují dvě varianty zápisu smyčky. Obě varianty začínají dvojicí klíčových slov *LOOP AT* následovanou neterminálním symbolem *user_defined*, který představuje název interní tabulky. V případě, že je při průchodu jeden záznam ukládán do proměnné, následuje klíčové slovo *INTO* a deklarace proměnné, tedy *DATA(user_defined)*. V opačném případě je záznam přiřazen ukazateli. Klíčové slovo *INTO* je nahrazeno klíčovým slovem *ASSIGNING* a deklarace proměnné je nahrazena deklarací ukazatele, *FIELDSYMBOL(user_defined)*. Příkaz je ukončen tečkou, terminálním symbolem DOT. Následuje blok vnořených příkazů. Příkaz je zastoupen neterminálním symbolem *statement*. V bloku příkazů se může vyskytovat libovolný počet příkazů. Smyčka LOOP je ukončena klíčovým slovem *ENDLOOP*.

```
loop_into :
  LOOP AT user_defined INTO DECLARATION_OP LPAREN user_defined
  RPAREN DOT statement* ENDLOOP DOT;

loop_assign :
  LOOP AT user_defined ASSIGNING FIELDSYMBOL LPAREN user_defined
  RPAREN DOT statement* ENDLOOP DOT;
```

5.1.10 Volání metod

Volání metod se skládá z názvu objektu, terminálního symbolu zastupujícího šipku, názvu metody a parametrů metody v kulatých závorkách. Název objektu i metody je zastoupen neterminálním symbolem *user_defined*. Parametry v závorkách představují přiřazení hodnoty proměnné, kterému předchází jedno ze tří klíčových slov. *IMPORTING* slouží k přiřazení výstupu metody k proměnné. *EXPORTING* se používá pro předání vstupu do metody. *CHANGING* pro změnu interní proměnné metody.

```
loop_into :
  LOOP AT user_defined INTO DECLARATION_OP LPAREN user_defined
  RPAREN DOT statement* ENDLOOP DOT;

loop_assign :
  LOOP AT user_defined ASSIGNING FIELDSYMBOL LPAREN user_defined
  RPAREN DOT statement* ENDLOOP DOT;
```

5.1.11 Čtení dat z interní tabulky

Data z interní tabulky je možné přiřadit do proměnné pomocí indexu viz. 5.1.7 nebo ukazatele. Přiřazení ukazatele je obdobné jako v případě smyčky LOOP, jen místo klíčových slov *LOOP AT* jsou použita klíčová slova *READ TABLE*. Název tabulky i ukazatele je opět zastoupen neterminálním symbolem *user_defined*.

```
read_assign :
   READ TABLE user_defined ASSIGNING FIELDSYMBOL LPAREN user_defined
   RPAREN DOT;
```

5.1.12 Čtení dat z databázové tabulky

Příkaz pro čtení z databázové tabulky má stejnou formu zápisu jako SQL. Kromě klíčových slov *SELECT*, *FROM*, *TABLE*, *INTO* a *WHERE* obsahuje pravidlo následující neterminální symboly. *User_defined* zastupující název databázové tabulky a interní tabulky, do které je výsledek uložen. *Select_param* představující kritéria výběru sloupců tabulky. *Where_param* není povinný a reprezentuje podmínky výběru řádků. Symbol *where_param* dále obsahuje neterminální symbol *where_op*, který představuje konkrétní podmínku výběru. Pokud je podmínek více, jsou spojeny klíčovými slovy *AND* nebo *OR*.

```
select_into :
  SELECT select_param FROM user_defined INTO TABLE DATA LPAREN
  user_defined RPAREN where_param? DOT;


select_param : ( '*' | user_defined (',' user_defined)* );
where_param : WHERE where_op ( ( AND | OR) where_op )*;
where_op : user_defined RELATIONAL_OP value;
```

5.1.13 Příkaz IF

Konstrukce příkazu IF začíná klíčovým slovem *IF*, kterému následuje neterminální symbol *condition* zastupující podmínku větvení a terminální symbol DOT, představující tečku. Následuje blok vnořených příkazů. Dále se může objevit libovolný počet neterminálních symbolů *st_else_if*, anebo maximálně jeden neterminální symbol *st_else*. Symbol *st_else_if* se skládá z klíčového slova *ELSEIF*, neterminálního symbolu *condition*, terminálního symbolu *DOT* a bloku vnořených příkazů. Symbol *st_else* se od *st_else_if* liší akorát absencí neterminálního symbolu *condition*. Podmínka větvení *condition* je složena z názvu proměnné, relačního operátoru a buď konkrétní hodnoty nebo názvu jiné proměnné. Konstrukce příkazu je ukončena klíčovým slovem *ENDIF*.

```
st_if : IF condition DOT statement* st_else_if* st_else? ENDIF DOT;


st_else_if : ELSEIF condition DOT statement*;
st_else : ELSE DOT statement*;
condition : user_defined RELATIONAL_OP ( user_defined | value );
```

5.1.14 Příkaz CASE

Příkaz CASE porovnává hodnotu požadované proměnné. Pravidlo se skládá z klíčového slova *CASE*, symbolu pro porovnávanou proměnnou. Za tečkou se libovolný počet neterminálních symbolů *st_case_when*. Pravidlo je ukončeno klíčovým slovem *ENDCASE*. Symbol *st_case_when* představuje jednotlivé větve příkazu. Symbol obsahuje klíčové slovo *WHEN*, konkrétní hodnotu nebo klíčové slovo *OTHERS* a blok vnořených příkazů.

```
st_case : CASE user_defined DOT st_case_when* ENDCASE DOT;
st_case_when : WHEN (value | OTHERS). statement*;
```

6 Návrh překladače

V této kapitole je popsána problematika vzájemné kompatibility verzí jazyka ABAP a důvod vzniku této diplomové práce. Následně je popsán návrh řešení uvedeného problému.

6.1 Úvod do problematiky

Jak bylo již zmíněno v kapitole 4.6 Verze jazyka ABAP nová verze jazyka ABAP 7.40 přinesla zásadní změny v syntaxi. Změny byly natolik zásadní a rozsáhlé, že již není nadále možné spouštět programy napsané ve verzi 7.40 na systémech SAP, kde je jazyk ABAP přítomen ve verzi 7.30 nebo starší.

Aktualizace systému SAP na novou verzi není ve většině firem možná okamžitě ze dvou důvodů. V menších firmách rozhoduje finanční hledisko - licence systému SAP jsou drahé [9]. Ve velkých podnicích bývá problémem zdlouhavý proces schválení nového software, kdy se musí otestovat, zda bude po aktualizaci vše fungovat, jak má.

Zdrojový kód napsaný pro verzi 7.40 lze přepsat tak, aby byl spustitelný i ve verzích starších. Nicméně tato úprava je velmi náročná na čas programátora, který musí zdrojový kód projít řádek po řádku a nekompatibilní příkazy přepsat do staré syntaxe.

Cílem je tedy minimalizovat náročnost na čas programátora a proces úpravy kódu co nejvíce zautomatizovat.

6.2 Návrh řešení

Výsledný program by měl umožnit jednoduše načíst zdrojový kód v poslední verzi jazyka ABAP určený k překladu. Výstupem programu bude opět zdrojový kód v jazyce ABAP, ale ve starší verzi. Program by měl uchovávat informace o samotném překladu. Například počet úspěšné převedených řádků nebo počet řádků, které je potřeba manuálně upravit, a to nejlépe i včetně informace o konkrétním místě ve zdrojovém kódu.

V ideálním případě by program načetl zdrojový kód přímo ze systému SAP. To bohužel není možné kvůli uzavřenosti systému SAP.

V úvahu tedy přichází načtení zdrojového kódu ze souboru na disku. Dalším způsobem načtení by mohlo být vložení zdrojového kódu do textového pole. Po načtení vstupního zdrojového kódu se program pokusí o jeho překlad, a to s využitím vytvořených gramatik, které jsou popsány v kapitole 5 Gramatika jazyka ABAP. Pokud se během překladu nepodaří nějaký řádek nebo část kódu přeložit, zaznamená program číslo řádku, kde k chybě došlo. Výstupem programu bude soubor s přeloženým zdrojovým kódem a soubor obsahující čísla řádků, na kterých došlo k chybě při překladu, nebo kde je nutný zásah programátora.

Načítání souboru bude realizováno pomocí jednoduchého grafického prostředí. Ve stejném okně bude možné zvolit výstupní adresář a název výstupního souboru. Dále bude možné zvolit i směr překladu, přičemž primární je překlad z verze 7.40 a novějších do verze 7.30 a starších. Opačný překlad by sloužil pouze ke zjednodušení a refaktoringu kódu.

7 Implementace překladače

V této kapitole jsou podrobně popsány obě realizace překladače. Přestože obě realizace překladače slouží ke stejnému účelu, jsou od sebe velmi odlišné. A to zejména v procesu překladu. Každý způsob realizace má svoje výhody i nevýhody.

7.1 Implementace s využitím nástrojů pro tvorbu překladačů

Tato varianta je realizována v jazyce JAVA využitím jednoho z nástrojů popsaných v kapitole 3 Nástroje pro tvorbu překladačů.

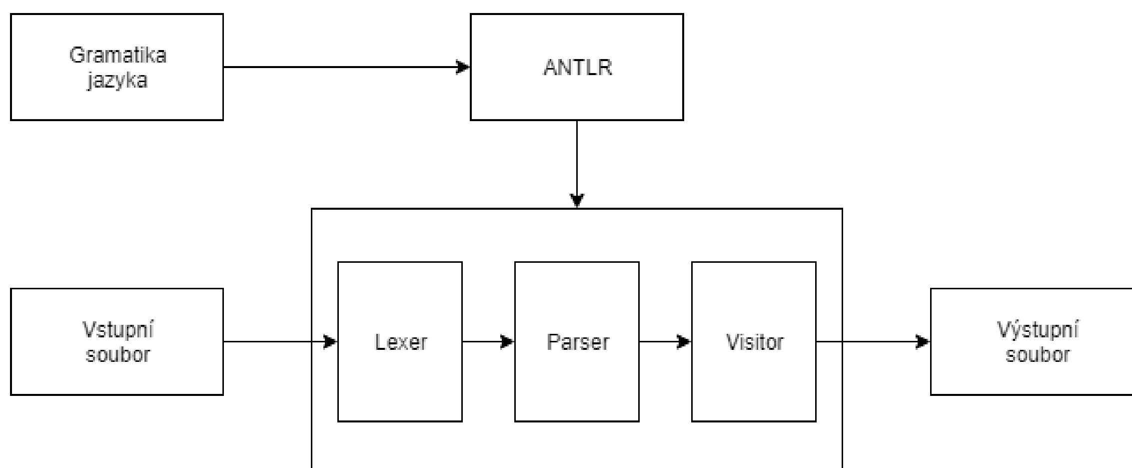
7.1.1 Výběr nástroje pro realizaci

Pro realizaci jsem nejdříve zvolil nástroj JavaCC, který se později ukázal jako nepoužitelný. Ačkoliv se podařilo na základě vytvořené gramatiky jazyka ABAP vygenerovat parser, nebylo možné jej následně spustit. Důvodem byly jednotlivé metody samotného parseru, jejichž délka přesahovala limity jazyka JAVA. Příčinou takto dlouhých metod byl veliký počet klíčových slov jazyka ABAP, kterých je dle dokumentace kolem tisíce. Redukce množiny klíčových slov pouze na ta používaná problém na první pohled vyřešila. Redukce proběhla na základě konzultace s konzultantem z firmy T-MC66. Bohužel se po doplnění zbývajících překladových pravidel problém s délkou metod objevil znovu.

Jako druhý byl zvolen nástroj ANTLR, v němž se z navržených gramatik podařilo vygenerovat parser a ten následně spustit bez jakýchkoliv chyb. Množina klíčových slov byla doplněna do plného rozsahu. Vygenerování i spuštění se obešlo bez chyb. Z toho vyplývá, že ANTLR generuje parsovací metody jiným způsobem ve srovnání s JavaCC.

7.1.2 Realizace řešení s vybraným nástrojem

Jelikož se pomocí JavaCC nepodařilo překladač zkompilovat, byl pro implementaci použit nástroj ANTLR. Nástroj z vytvořené gramatiky při kompilaci automaticky vygeneroval několik souborů. Mezi tyto soubory patří například seznam tokenů, lexer, parser a v neposlední řadě soubor definující rozhraní takzvaného visitora. Ten slouží pro procházení stromu a generování vstupního kódu.

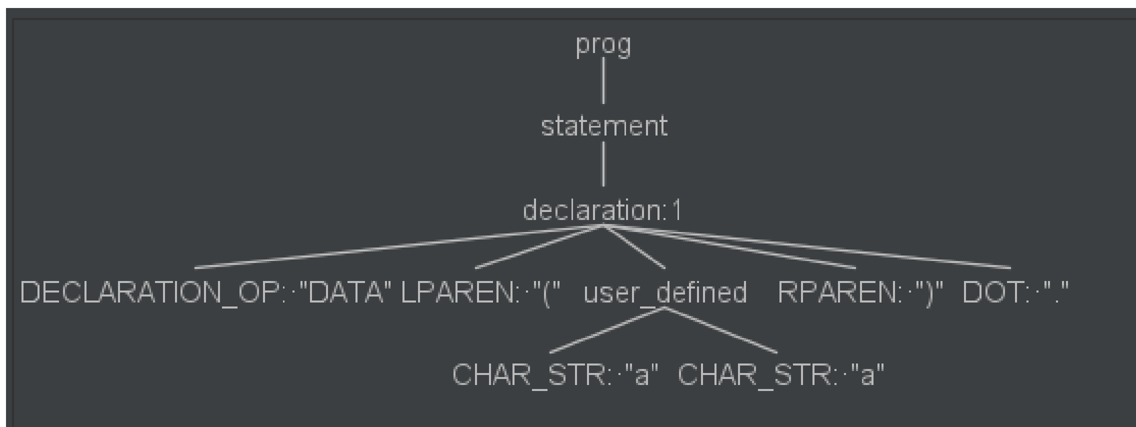


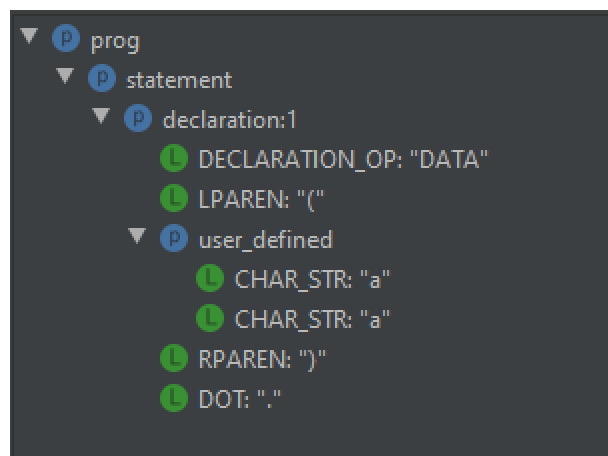
Obrázek 5 - Schéma překladače.

Gramatika jazyka

Gramatiku pro verzi jazyka ABAP ve verzi 7.40 a vyšší bylo potřeba z původní EBNF formy upravit tak, aby její zápis odpovídal konvencím ANTLR. Mezi tyto konvence patří například pojmenovaní jednotlivých terminálních a neterminálních symbolů. Neterminální symboly jsou nyní pojmenovány malými písmeny, terminální symboly pak velkými písmeny. Všechny pevně definované textové řetězce, které byly součástí překladových pravidel byly nahrazeny odpovídajícími terminálními symboly. Po provedení těchto úprav proběhla kompilace gramatiky a vygenerování překladače bez jakýchkoliv varování nebo chyb.

Před samotnou implementací překladače bylo nutné nejprve navrženou gramatiku otestovat. Díky zvolenému vývojovému prostředí, IDEA od JetBrains s nainstalovaným doplňkem pro ANTLR, bylo testování jednotlivých pravidel velmi intuitivní. Zpětnou vazbou testování byl dynamicky generovaný derivační strom a detailní popis chyb v případě, že se v testovacím řetězci objevil nějaký nečekaný symbol.





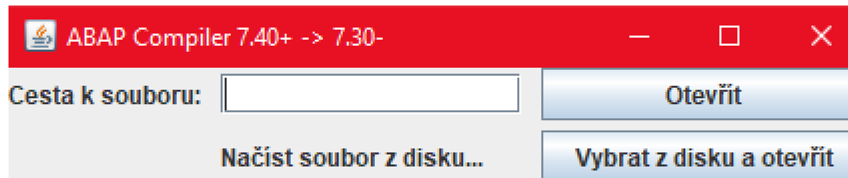
Obrázek 6 – Ukázka derivačního stromu pravidla.

Na základě tohoto testování bylo nutné některá pravidla upravit. Testování například odhalilo vícenásobné definice terminálních symbolů. V současném stavu gramatika akceptuje zdrojový kód v jazyce ABAP verze 7.40 a vyšší. Nicméně se gramatika bude s přibývajícími testy na reálných zdrojových kódech používaných v praxi dále vyvíjet a zdokonalovat.

Vstup a výstup programu

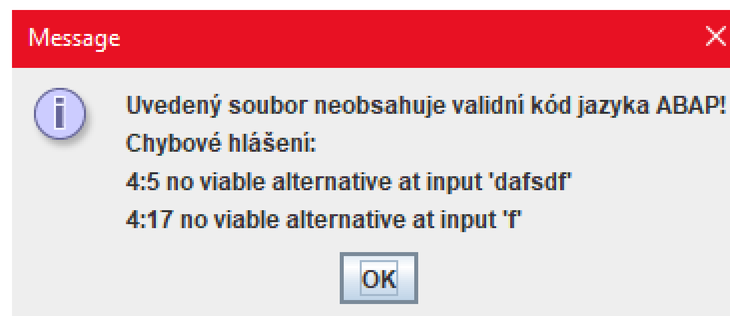
Vstupem a výstupem programu je textový soubor obsahující kód jazyka ABAP. Vstupní soubor, obsahující zdrojový kód jazyka ABAP ve verzi 7.40 a vyšší, lze uživatelsky zvolit pomocí jednoduchého grafického rozhraní. Uživateli jsou k dispozici dva způsoby volby načtení souboru. První možností je zadání absolutní cesty k souboru. Překlad je spuštěn po kliknutí na tlačítko „Otevřít“. V případě, že uživatel zadá cestu k neexistujícímu souboru, je o tom patřičně informován vyskakovacím oknem. Druhá možnost umožňuje soubor vybrat ze souborového systému. Po stisknutí tlačítka „Open“, v okně pro výběr souboru, je spuštěn překlad vybraného souboru. Výstupní soubor se nachází ve stejné složce jako uživatelsky zvolený vstupní soubor. Název výstupního soboru je generován

programově, ve formátu: *rrrrmmdd_hhmmss_ABAP_compiler_output.txt*, kde část řetězce *rrrrmmdd_hhmmss* představuje údaj o datu a čase spuštění překladu.



Obrázek 7 - Grafické rozhraní překladače.

V obou případech je uživatel informován o výsledku překladu. V případě úspěšného překladu je uživateli ve vyskakovacím okně zobrazena absolutní cesta k přeloženému souboru. V případě, že se během překladu vyskytly chyby, je jejich seznam, včetně konkrétní pozice, zobrazen uživateli ve vyskakovacím okně.



Obrázek 8 - Hlášení o neúspěšném překladu.

Princip překladu a generování výstupního kódu

Načtený zdrojový kód je uchovaný v proměnné typu *CharStream*, její obsah lze chápat jako proud jednotlivých znaků. Znakový proud je následně předán lexikálnímu analyzátoru. Funkce lexikálního analyzátoru je popsána v kapitole 2.2 Lexikální analýza. Výsledek lexikální analýzy je uchován v proměnné typu *CommonTokenStream*, tedy proud jednotlivých tokenů. Proud tokenů je předán syntaktickému analyzátoru. Popis funkce syntaktického analyzátoru je popsán v kapitole 2.3 Syntaktická analýza. Výstupem syntaktické analýzy je derivační strom. Derivační strom je následně procházen tzv. *Visitorem*.

Každému pravidlu gramatiky odpovídá jedna metoda *Visitoru*. Název těchto metod je tvořen slovem „visit“ a názvem pravidla gramatiky. Například metoda *visitDeclaration* je volána, pokud aktuální uzel stromu odpovídá pravidlu *Declaration*, *visitAssignment* je volána, pokud se jedná o příkaz přiřazení. Ostatní metody jsou pojmenovány analogicky.

Pravidla gramatiky lze, na základě jejich složitosti, rozdělit na dvě skupiny. Za jednoduchá pravidla lze pokládat ta pravidla, která obsahují pouze terminální symboly, nebo jejich struktura není příliš složitá. Jako jednoduché pravidlo lze považovat pravidlo pro uživatelsky definované názvy proměnných. Složitým pravidlem je myšleno takové pravidlo, které obsahuje kterékoliv jiné pravidlo gramatiky. Jedním ze složitých pravidel je například pravidlo pro smyčku LOOP nebo konstrukci IF, kdy se kromě terminálních symbolů vyskytují i symboly neterminální. V případě jednoduchého převodového pravidla je možné pro získání textu uzlu, který pravidlu odpovídá, použít vestavěnou metodu *getText()*. V opačném případě, odpovídá-li uzel složitému pravidlu, není použití metody *getText()* možné. Proto bylo nutné pro všechna složitá pravidla vytvořit odpovídající třídy s přetíženou metodou *toString()*.

Vytvořené třídy mají zpravidla dva a více atributů, v případě, že má třída pouze jeden atribut, je tento atribut typu *List<String>*. Tyto atributy slouží pro uchování názvů proměnných, podmínek, vnořených příkazů a podobně. Zdrojový kód níže odpovídá deklaraci třídy pro převodové pravidlo *Declaration*. Atribut *variable_name* uchovává název uživatelský definované proměnné. Druhý atribut *variable_type* obsahuje konkrétní datový typ.

```
package RuleClasses;

public class Declaration implements IStatement{
    public String variable_name;
    public String variable_type;

    public Declaration(String _variable_name, String _variable_type){
        this.variable_name = _variable_name;
        this.variable_type = _variable_type;
    }

    @Override
    public String toString() {
        return String.format("data %s type %s.", variable_name, variable_type);
    }
}
```

Zdrojový kód 10 - Třída pravidla Declaration.

Následující ukázka zdrojového kódu obsahuje kód metody *visitDeclaration*. Proměnná *children_list* obsahuje seznam všech potomků příkazu deklarace. Získání seznamu potomků obstarává metoda *get_children_list* popsaná níže. Název proměnné se v seznamu *children_list* nachází na pozici s indexem 0. Jelikož v jazyce ABAP lze od verze 7.40 deklarovat proměnné i bez určení konkrétního datového typu, je přiřazení to

proměnné *variable_type* ošetřeno klauzulí try-catch. V případě, že se na pozici s indexem 1 nenachází datový typ, je nahrazen textovým řetězcem „!!N/A“. Návratovou hodnotou je nová instance třídy *Declaration*.

```
@Override public T visitDeclaration(abap_newParser.DeclarationContext ctx) {
   List<String> children_list = get_children_list(ctx);
   String variable_name = children_list.get(0);
   String variable_type;

   try{
      variable_type = children_list.get(1);
   }
   catch (Exception e){
      variable_type = "!!N/A";
   }

   return (T) new Declaration(variable_name, variable_type);
}
```

Zdrojový kód 11 - Metoda Visitoru – visitDecalariotn.

Implementace ostatních metod i deklarace tříd je velmi obdobná, proto není nutné je všechny detailně popisovat. Kompletní zdrojový kód *Visitoru* a deklarace tříd je k nalezení v přílohách této práce.

Jak již bylo zmíněno dříve, pro získání textu uzlu odpovídajícímu složitému překladovému pravidlu není možné použít vestavěnou metodu *getText*, bylo tedy nutné naimplementovat metodu vlastní. Ukázka zdrojového kódu níže obsahuje kód metody *get_children_list*, která slouží k získání seznamu všech potomků požadovaného uzlu. Jelikož každé pravidlo má vlastní visit metodu, je nutné jednotlivá pravidla od sebe odlišit. Každé pravidlo má v parseru svůj kontext, například: *Declaration* – *DeclarationContext*, *Assignment* – *AssignmentContext*, atd. Zavolání správné metody zaručuje jednoduché porovnání třídy potomka s třídou kontextu, např. *child.getClass() == parser.DeclarationContext.class*. Jedná-li se o uživatelsky definovanou proměnnou, datový typ nebo operátor je do seznamu potomků přidána návratová hodnota metody *getText*. Pokud se nejedná o žádný ze tří uvedených a ani o terminální symbol, je potřeba s potomkem pracovat jako s objektem. Objekt potomka, konkrétně jeho kontext, vrací metoda *getPayload*. Následně je na základě třídy potomka volána příslušná visit metoda. Do seznamu *children_list* je přidána textová reprezentace objektu daného uzlu.

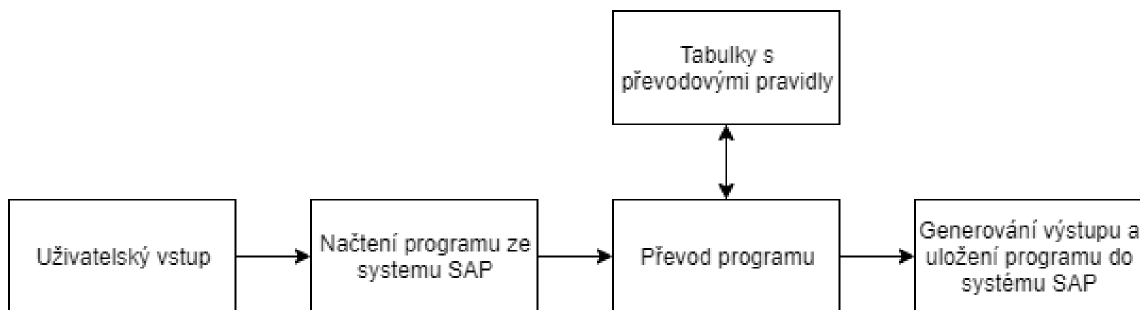
```
public List<String> get_children_list(ParserRuleContext ctx){
   List<String> children_list = new ArrayList<>();
   for(int i = 0; i < ctx.getChildCount(); i++){
      ParseTree child = ctx.getChild(i);

      if (child.getClass() == abap_newParser.User_definedContext.class ||
         child.getClass() == abap_newParser.Data_typeContext.class ||
         child.getClass() == abap_newParser.OperatorContext.class)
      {
         children_list.add(child.getText());
      }
      else if(child.getClass() != TerminalNodeImpl.class){
         Object child_ctx = child.getPayload();
         // child class detection
         if(child.getClass() == abap_newParser.DeclarationContext.class)
           children_list
           .add(visitDeclaration((abap_newParser.DeclarationContext) child_ctx)
           .toString());
         else if(child.getClass() == abap_newParser.StatementContext.class){
            List<String> temp_children_list =
             get_children_list((ParserRuleContext) child_ctx);
            String temp_children_list_str = "";
            for(String temp_child : temp_children_list){
               if (temp_children_list_str == "")
                  temp_children_list_str = temp_child;
               else temp_children_list_str += String.format(";%s", temp_child);
            }
            children_list.add(temp_children_list_str);
         }
         else if(child.getClass() == abap_newParser.St_ifContext.class)
           children_list
             .add(visitSt_if((abap_newParser.St_ifContext) child_ctx).toString());
         .
         .
      }
   }
   return children_list;
}
```

Zdrojový kód 12 - Metoda get_children_list.

7.2 Implementace v systému SAP

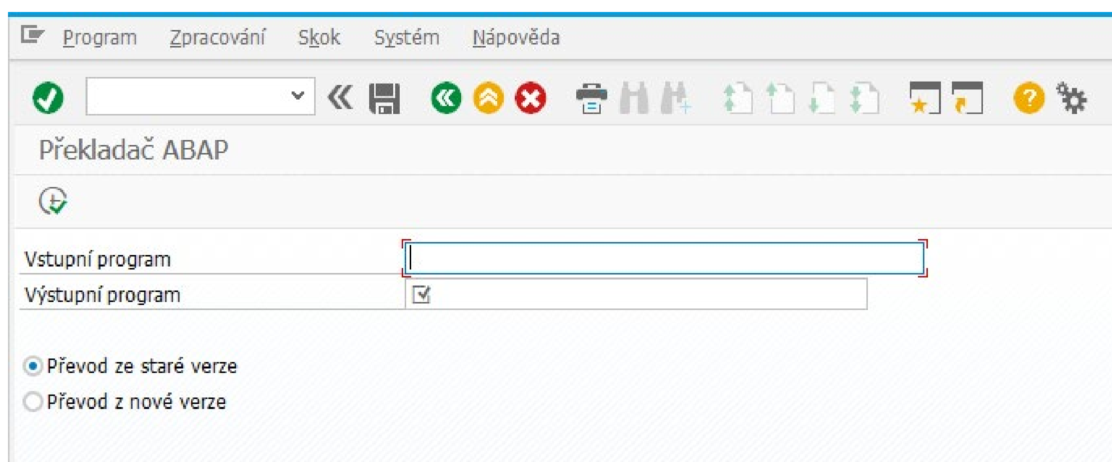
Tato varianta vznikla na základě požadavku ve firmě, která zadala tuto diplomovou práci. Hlavním rozdílem oproti řešení pomocí ANTLR je to, že se nejedná o překladač jako takový. Není zde přítomen lexikální, syntaktický ani sémantický analyzátor. Gramatiky jazyka jsou nahrazeny tabulkami obsahující pravidla pro převod. Pravidla v tabulkách ovšem z vytvořených gramatik vychází.



Obrázek 9 - Schéma převodního programu.

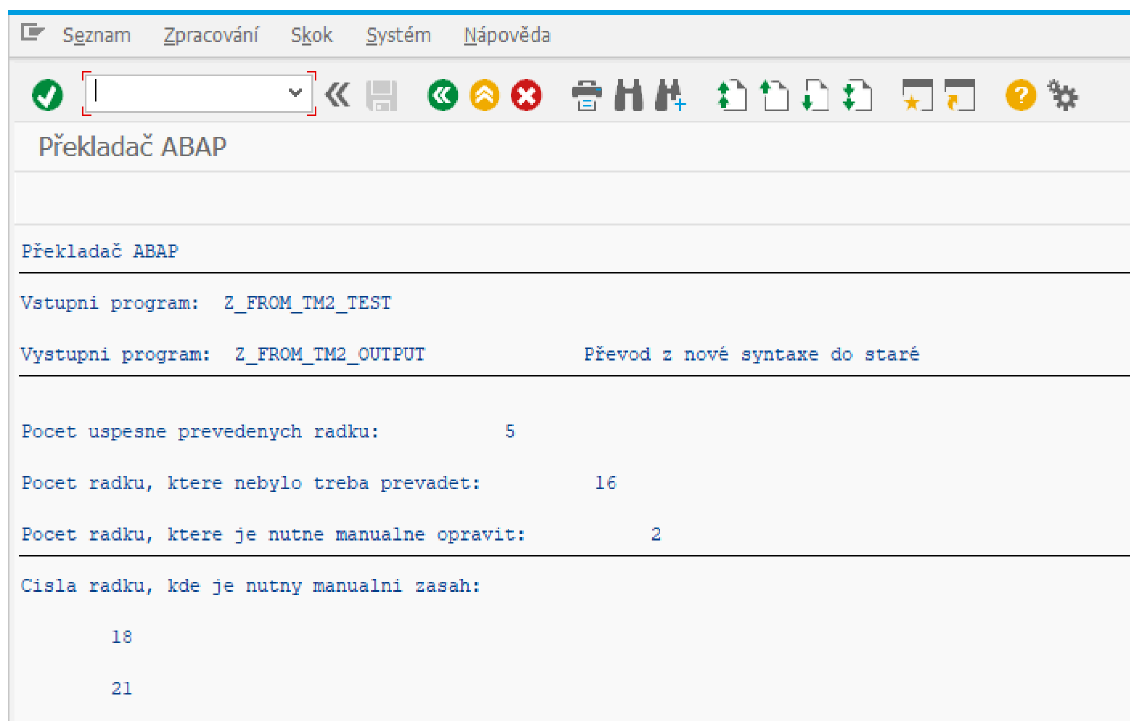
7.2.1 Vstup a výstup programu

Po spuštění je uživateli zobrazena takzvaná výběrová obrazovka. Pomocí této obrazovky jsou programu předány všechny potřebné parametry. Uživatel zvolí název programu, který chce převést, dále zvolí název výstupního programu a na závěr zvolí směr převodu syntaxe.



Obrázek 10 - Grafické rozhraní programu v systému SAP.

Výstupem je nový program s názvem, který uživatel zvolil. Program obsahuje převedený zdrojový kód. Po dokončení převodu se uživateli zobrazí jednoduchá obrazovka obsahující informace o průběhu převodu. Uživateli je zobrazen počet úspěšně převedených řádků, počet řádků, kde je nutné manuální zásah programátora, a to včetně čísel řádků. Poslední zobrazenou informací je počet řádků, které nepodléhají převodovým pravidlům, to znamená, že jejich zápis je v obou verzích jazyka totožný.



Obrázek 11 - Výstup programu v systému SAP.

7.2.2 Tabulky s pravidly

Pro převod jednotlivých příkazů využívá program tabulku s pravidly. Program má k dispozici dvě tabulky. Jednu pro verzi 7.40 a vyšší, druhou pro 7.30 a nižší. Obě tabulky mají následující strukturu:

Transp.tabulka ZABAPGRAMMARNEW2 Aktiv.
Krátký popis Tabulka s pravidly gramatiky

Vlastnosti | Expedice a údržba | Pole | Nápověda/kontrola zadávání

Náp.vyhl. Instal

Pole	Key	Ini...	Datový prvek	Datový typ	Délka
MANDT	✓	✓	MANDT	CLNT	3
KLEFT	✓	✓	ZKEY	CHAR	119
KRIGHT		✓	ZRULE	CHAR	255
GROUPS			ZGROUPS	CHAR	11
REGEX			ZREGEX	CHAR	255

Obrázek 12 - schéma tabulky pro uchování převodových pravidel

Pole MANDT je pole systému SAP, obsahuje ID klienta, který může k tabulce přistupovat a tabulku následně upravovat. Pokud se toto pole v tabulce nenachází, tabulka

se stává nezávislou na klientovi. Pole KLEFT obsahuje identifikátor převodového pravidla. Aby byl převod možný, musí být množina prvků KLEFT totožná v obou tabulkách. Pole KRIGHT slouží pro uchování převodového pravidla. Poslední dvě položky GROUPS a REGEX obsahují informace pro umožnění převodu. GROUPS obsahuje jednotlivé shody v regulárním výrazu. Jedná se o řetězec, ve kterém jsou čísla skupin oddělené středníkem. Čísla skupin jsou seřazená tak, jak se vyskytují v novém pravidle. Pole REGEX ukrývá regulární výraz sloužící pro separaci proměnných z příkazu. V následující tabulce je zobrazen příklad obsahu tabulky pro jedno vybrané pravidlo.

Tabulka 17 - Příklad převodového pravidla.

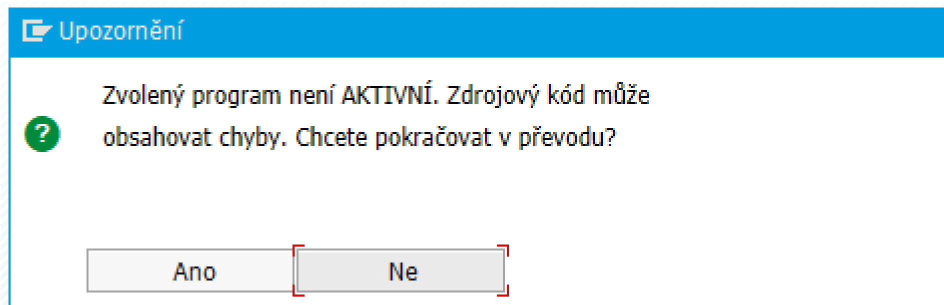
Pole	Obsah
MANDT	001
KLEFT	DECLARATION
KRIGHT	DATA\(([a-zA-Z0-9_]+)\)\.
GROUPS	1;0
REGEX	([a-zA-Z0-9_]+)

7.2.3 Princip převodu

Načtení a zpracování zdrojového kódu

Před samotným načtením dojde ke kontrole zdrojového kódu požadovaného programu. Tento krok je nutné udělat, aby se minimalizovala pravděpodobnost překladu nespustitelného nebo jinak nevalidního kódu. Aby bylo možné v systému SAP spustit uživatelský program, musí být daný program nejprve „aktivní“. Aktivovat lze pouze ty programy, které neobsahují chyby v syntaxi. Tento proces ve své podstatě může být přirovnán k sestavení zdrojového kódu v jiném programovacím jazyce, jako například JAVA, C a podobně. Informace o tom, zda je daný program aktivní se nachází v systémové tabulce REPOSRC, konkrétně ve sloupci R3STATE. V tomto sloupci se mohou vyskytovat buď hodnoty ‚A‘ – aktivní, nebo ‚I‘ – neaktivní. Stav aktivace programu lze získat pomocí příkazu SELECT. Konkrétně v této podobě: SELECT R3STATE FROM REPOSRC INTO it_reposrc WHERE PROGNAME = template ORDER BY UDAT DESCENDING UTIME DESCENDING. Klauzule ORDER BY je

použita, protože v tabulce se pro zvolený program může vyskytovat více záznamů. Nicméně důležitý je pouze ten nejnovější. Výsledek dotazu je poté vyhodnocen jednoduchou podmínkou. Pokud je výsledkem dotazu ‚I', zobrazí se uživateli okno s upozorněním, že požadovaný program není aktivní, a proto může obsahovat chyby. Pokud je zvolený program aktivní, nebo se uživatel rozhodl pokračovat v převodu i s neaktivním programem, dojde k načtení zdrojového kódu do proměnné.



Obrázek 13 - Upozornění na neaktivní program.

Pro uchování zdrojového kódu slouží interní tabulka, jejíž řádky jsou datového typu STRING. Každý řádek tabulky pak odpovídá jednomu řádku zdrojového kódu. Načtení zdrojového kódu se provádí pomocí příkazu READ REPORT <název programu> INTO <interní tabulka>. Následně je interní tabulka se zdrojovým kódem dvakrát iterována pomocí smyčky LOOP. Jelikož je možné příkazy jazyka ABAP rozdělit na více řádků, nemusí každý řádek končit tečkou. Proto při prvním průchodu tabulkou takto rozdělené příkazy spojeny do jednoho řádku, aby bylo možné je při druhém průchodu převést využitím některého z převodových pravidel. V druhé iteraci probíhá samotný převod.

Převod jednotlivých řádků

Před samotným pokusem o převod je provedena jednoduchá kontrola, o jaký typ příkazu se jedná. Program rozlišuje tři skupiny příkazů. První skupinou jsou uživatelské komentáře zdrojového kódu. V jazyce ABAP komentáře začínají znakem hvězdičky (*) nebo uvozovkou ("). Druhá skupina označuje všechny příkazy, které začínají některým z klíčových slov jazyka ABAP. Všechna klíčová slova jsou uchována v tabulce. Posledním typem příkazu je příkaz přiřazení. Uživatelské komentáře a příkazy přiřazení nepodléhají převodu. Tedy kromě případu, kdy je proměnné přiřazena konkrétní hodnota tabulky pomocí indexu.

Komentáře zdrojového kódu není třeba nijak převádět. V případě, že se jedná o příkaz přiřazení dojde k pokusu o převod pomocí pravidla pro index tabulky. Pokud řádek

vyhovuje regulárnímu výrazu, dojde k jeho převodu. Není-li nalezena shoda s regulárním výrazem, jedná se o běžné přiřazení hodnoty proměnné, které není třeba převádět. V případě příkazu jazyka ABAP jsou postupně zkoušena všechna převodová pravidla z tabulky pravidel. Je-li nalezena shoda je použito odpovídající pravidlo z tabulky pravidel cílové verze.

Aby bylo možné zachovat názvy proměnných, datové typy či jiné parametry jednotlivých příkazů, jsou využity skupiny v regulárním výrazu. Jazyk ABAP umožňuje rozlišit až 6 skupin v rámci jednoho regulárního výrazu. Správné pořadí nalezených skupin zajišťuje řetězec ve sloupci GROUPS v tabulce pravidel. Jednotlivé skupiny jsou do nového příkazu přiřazeny pomocí příkazu REPLACE. Nahrazovaným řetězcem je řetězec ze sloupce REGEX v tabulce pravidel. Nahrazení je realizováno pomocí iterace nad interní tabulkou obsahující čísla jednotlivých skupin. Tato tabulka je vytvořena voláním příkazu SPLIT nad řetězcem GROUPS, kde oddělovačem je středník. Pokud byly všechny výskyty řetězce REGEX úspěšně nahrazeny dojde k navýšení počítadla úspěšně převedených řádků *conv_success*. Pokud obsahuje řetězec GROUPS číslo 0, není možné aktuální řádek kompletně převést. Na pozici odpovídající této „skupině“ je vložen řetězec *„doplnit_ručně“*. Je navýšeno počítadlo nepřeveditelných řádků *conv_fail* a číslo řádku, na kterém chyba vznikla je uloženo do interní tabulky *failed_lines*, aby bylo možné chybějící argument později ručně doplnit. Výsledek převodu je uložen do interní tabulky *it_out_src*. Kromě převedeného příkazu je formě komentáře zachován i původní řádek.

```
"puvodni prikaz - LOOP AT S_CLIENT INTO DATA(CLNT).
DATA CLNT LIKE LINE OF S_CLIENT. LOOP AT S_CLIENT INTO CLNT.
...
```

Zdrojový kód 13 - Příklad převedeného řádku.

Generování převedeného programu

Poslední fází převodu je vytvoření nového programu v systému SAP s převedeným zdrojovým kódem. Vytvoření výsledného programu je provedeno pomocí příkazů:

```
insert report out_name from it_out_src.
commit work.
```

Zdrojový kód 14 - Uložení převedeného programu.

První příkaz slouží k vytvoření programu se jménem uchovaným v proměnné *out_name* a vložení zdrojového kódu nacházejícího se v interní tabulce *it_out_src*. Druhý příkaz

uloží provedené změny v systému. Takto vytvořený program lze následné spustit či upravit v transakci SE38.

8 Závěr

V rámci diplomové práce byla napsána gramatika pro jazyk ABAP 7.40 a vyšší. Na základě vytvořené gramatiky byl navržen a realizován překladač. Překladač slouží k překladu zdrojového kódu jazyka ABAP z verze 7.40 a vyšší na verzi 7.30 a nižší.

Překladač byl realizován v jazyce JAVA s využitím nástroje ANTLR. Překladač na vstupu načte textový soubor obsahující zdrojový kód v jazyce ABAP. Zdrojový kód je následně překladačem zpracován. Výstupem je textový soubor obsahující ekvivalentní zdrojový kód v jazyce ABAP odpovídajícím verzi 7.30 a starší. Komunikaci s uživatelem zajišťuje jednoduché grafické rozhraní umožňující výběr textového souboru uloženého na disku počítače. O výsledku překladu je uživatel informován, včetně popisu případných chyb vzniklých při překladu. V rámci této diplomové práce bylo kromě samotného překladače realizováno řešení v jazyce ABAP spustitelné přímo v systému SAP. V tomto případě se však nejedná o překladač jako takový. Toto řešení bylo implementováno a testováno na systému společnosti T-MC66.

Do budoucna by mohla být práce rozšířena o překlad opačným směrem, tedy z verze 7.30 na verzi novější. Účelem tohoto směru překladu by byl spíše refaktoring a zjednodušení kódu. Pro realizaci by bylo nutné upravit a otestovat gramatiku jazyka ABAP 7.30, jejíž základní podoba byla v rámci této diplomové práce také navržena. Kromě gramatiky by bylo nutné znovu vygenerovat *Lexer* a *Parser* a znovu naimplementovat *Visitora*.

9 Seznam použité literatury

[1] Čejka M., Hruška T., Beneš M.: Překladače, Učební text VUT Brno.

[2] Wirth N.: Compiler Construction, Addison-Wesley, 1996, ISBN 0-201-40353-6.

[3] Parr T.: The Definitive ANTLR Reference, The Pragmatic Bookshelf, 2007, ISBN: 0-9787392-5-6.

[4] Bandari K.: Complete ABAP, SAP Press, 2017, ISBN 978-1-4932-1273-6.

[5] Aho A., Lam M., Sethi R., Ullman J.: Compilers: Principles, Techniques, and Tools, Pearson Education, 2006, ISBN: 0-201-10088-6.

[6] Zalavadia B.: Compiler Construction Tools, IJSRD: International Journal for Scientific Research & Development, 2014, 2014(1), ISSN: 2321-0613.

[7] ABAP Keyword Documentation [online]. Dostupné z: https://help.sap.com/doc/PRODUCTION/abapdocu_751_index_htm/7.51/en-US/abenabap.htm

[8] ABAP 7.40 Quick Reference [online]. Dostupné z: https://blogs.sap.com/2015/10/25/abap-740-quick-reference/

[9] SAP Pricing Estimate [online]. Dostupné z: https://www.sap.com/products/cloud-platform/pricing/estimator-tool.html

[10] JavaCC Documentation [online]. Dostupné z: https://javacc.github.io/javacc/

[11] Donelly Ch., Stallman R.: GNU Bison, Free Software Foundation, 2019, ISBN 1-882114-44-2. Dostupné z: https://www.gnu.org/software/bison/manual/bison.pdf

[12] The Lex & Yacc Page [online]. Dostupné také z: http://dinosaur.compilertools.net/

[13] ABAP. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-. Dostupné z: https://en.wikipedia.org/wiki/ABAP

10 Přílohy

Příloha A – zdrojový kód Visitoru překladače

```
// Generated from C:/Users/honva/IdeaProjects/DP_ABAP_compiler\abap_new.g4 by ANTLR
4.7.2
import RuleClasses.*;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.*;

import java.util.ArrayList;
import java.util.List;


public class abap_newBaseVisitor<T> extends AbstractParseTreeVisitor<T> implements
abap_newVisitor<T> {
   public static List<String> compiler_output = new ArrayList<>();

   @Override public T visitProg(abap_newParser.ProgContext ctx) {
      return visitChildren(ctx);
   }

   @Override public T visitStatement(abap_newParser.StatementContext ctx) {
      //if (ctx.depth() == 2) System.out.println(visitChildren(ctx));
      if (ctx.depth() == 2) compiler_output.add(visitChildren(ctx).toString());
      return null;
   }

   @Override public T visitGeneric(abap_newParser.GenericContext ctx) {
      List<String> statement_parts = get_children_list(ctx);
      return (T) new Generic(statement_parts);
   }

   @Override public T visitKeyword(abap_newParser.KeywordContext ctx) {
      return (T) ctx;
   }

   @Override public T visitUser_defined(abap_newParser.User_definedContext ctx) {
      return (T) ctx;
   }

   @Override public T visitOperand(abap_newParser.OperandContext ctx) { return (T)
ctx; }

   @Override public T visitOperator(abap_newParser.OperatorContext ctx) { return (T)
ctx; }

   @Override public T visitData_type(abap_newParser.Data_typeContext ctx) { return
(T) ctx; }

   @Override public T visitAddition(abap_newParser.AdditionContext ctx) { return (T)
ctx; }

   @Override public T visitDeclaration(abap_newParser.DeclarationContext ctx) {
      List<String> children_list = get_children_list(ctx);
      String variable_name = children_list.get(0);
      String variable_type;

      try{
         variable_type = children_list.get(1);
      }
```

```
        catch (Exception e){
            variable_type = "!!N/A";
        }

        return (T) new Declaration(variable_name, variable_type);
    }

    @Override public T visitValue(abap_newParser.ValueContext ctx) { return
visitChildren(ctx); }

    @Override public T visitNumeric(abap_newParser.NumericContext ctx) { return (T)
ctx.getText(); }

    @Override public T visitMath_expression(abap_newParser.Math_expressionContext ctx)
{
        return visitChildren(ctx);
    }

    @Override public T visitT(abap_newParser.TContext ctx) { return
visitChildren(ctx); }

    @Override public T visitF(abap_newParser.FContext ctx) { return
visitChildren(ctx); }

    @Override public T visitAssignment(abap_newParser.AssignmentContext ctx) {
        List<String> children_list = get_children_list(ctx);
        return (T) new Assignment(children_list.get(0), children_list.get(1));
    }

    @Override public T visitLoop_into(abap_newParser.Loop_intoContext ctx) {
        List<String> children_list = get_children_list(ctx);
        String table_name = children_list.get(0);
        String wa_name = children_list.get(1);
        List<String> nested_statements = new ArrayList<>();
        for(int i = 2; i < children_list.size(); i++){
            nested_statements.add(children_list.get(i));
        }
        return (T) new LoopInto(table_name, wa_name, nested_statements);
    }

    @Override public T visitLoop_assign(abap_newParser.Loop_assignContext ctx) {
        List<String> children_list = get_children_list(ctx);
        String table_name = children_list.get(0);
        String field_symbol_name = children_list.get(1);
        List<String> nested_statements = new ArrayList<>();
        for(int i = 2; i < children_list.size(); i++){
            nested_statements.add(children_list.get(i));
        }
        return (T) new LoopAssign(table_name, field_symbol_name, nested_statements);
    }

    @Override public T visitMethod_call(abap_newParser.Method_callContext ctx) {
        List<String> children_list = get_children_list(ctx);
        String object_name = children_list.get(0);
        String method_name = children_list.get(1);
        List<String> nested_statements = new ArrayList<>();
        for(int i = 2; i < children_list.size(); i++){
            nested_statements.add(children_list.get(i));
        }
        return (T) new MethodCall(object_name, method_name, nested_statements);
    }

    @Override public T visitRead_assign(abap_newParser.Read_assignContext ctx) {
        List<String> children_list = get_children_list(ctx);
        String fs_name = children_list.get(1);
        String table_name = children_list.get(0);
```

```
    return (T) new ReadAssign(fs_name, table_name);
}

@Override public T visitSelect_into(abap_newParser.Select_intoContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String table_name = children_list.get(1);
    String wa_name = children_list.get(2);
    String select_param = children_list.get(0);
    String where_param;
    try {
        where_param = children_list.get(3);
    }
    catch (Exception e){
        where_param = "";
    }

    return (T) new SelectInto(select_param, table_name, wa_name, where_param);
}

@Override public T visitSelect_param(abap_newParser.Select_paramContext ctx) {
    return (T) ctx.getText();
}

@Override public T visitWhere_param(abap_newParser.Where_paramContext ctx) {
    return (T) ctx.getText();
}

@Override public T visitWhere_op(abap_newParser.Where_opContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String variable_name = children_list.get(0);
    String variable_value = children_list.get(1);
    String relation_operator = ctx.getChild(1).getText();
    return (T) new WhereOp(variable_name, relation_operator, variable_value);
}

@Override public T visitSelect_single(abap_newParser.Select_singleContext ctx) {
    return visitChildren(ctx);
}

@Override public T visitTable_index(abap_newParser.Table_indexContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String table_name = children_list.get(1);
    String variable_name = children_list.get(0);
    String table_index = ctx.getChild(4).getText();
    return (T) new TableIndex(table_name, variable_name, table_index);
}

@Override public T visitSt_if(abap_newParser.St_ifContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String condition = children_list.get(0);
    List<String> nested_statements = new ArrayList<>();
    for(int i = 1; i < children_list.size(); i++){
        nested_statements.add(children_list.get(i));
    }
    return (T) new If(condition, nested_statements);
}

@Override public T visitSt_else_if(abap_newParser.St_else_ifContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String condition = children_list.get(0);
    List<String> nested_statements = new ArrayList<>();
    for(int i = 1; i < children_list.size(); i++){
        nested_statements.add(children_list.get(i));
    }
    return (T) new ElseIf(condition, nested_statements);
```

```
}

@Override public T visitSt_else(abap_newParser.St_elseContext ctx) {
    List<String> children_list = get_children_list(ctx);
    List<String> nested_statements = new ArrayList<>();
    for(int i = 0; i < children_list.size(); i++){
        nested_statements.add(children_list.get(i));
    }
    return (T) new Else(nested_statements);
}

@Override public T visitCondition(abap_newParser.ConditionContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String variable_name = children_list.get(0);
    String relation_operator = ctx.getChild(1).getText();
    String variable_value = children_list.get(1);
    return (T) new Condition(variable_name, relation_operator, variable_value);
}

@Override public T visitSt_case(abap_newParser.St_caseContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String variable_name = children_list.get(0);
    List<String> nested_statements = new ArrayList<>();
    for(int i = 1; i < children_list.size(); i++){
        nested_statements.add(children_list.get(i));
    }
    return (T) new Case(variable_name, nested_statements);
}
@Override public T visitSt_case_when(abap_newParser.St_case_whenContext ctx) {
    List<String> children_list = get_children_list(ctx);
    String when_value = children_list.get(0);
    List<String> nested_statements = new ArrayList<>();
    for(int i = 1; i < children_list.size(); i++){
        nested_statements.add(children_list.get(i));
    }
    return (T) new CaseWhen(when_value, nested_statements);
}

@Override public T visitComment(abap_newParser.CommentContext ctx) { return
visitChildren(ctx); }

// helper methods
public List<String> get_children_list(ParserRuleContext ctx){
    List<String> children_list = new ArrayList<>();
    for(int i = 0; i < ctx.getChildCount(); i++){
        ParseTree child = ctx.getChild(i);
        if (child.getClass() == abap_newParser.User_definedContext.class ||
child.getClass() == abap_newParser.Data_typeContext.class || child.getClass() ==
abap_newParser.OperatorContext.class){
            children_list.add(child.getText());
        }
        else if(child.getClass() != TerminalNodeImpl.class){
            Object child_ctx = child.getPayload();
            // child class detection
            if(child.getClass() == abap_newParser.DeclarationContext.class)
children_list.add(visitDeclaration((abap_newParser.DeclarationContext)
child_ctx).toString());
            else if(child.getClass() == abap_newParser.StatementContext.class){
                List<String> temp_children_list =
get_children_list((ParserRuleContext) child_ctx);
                String temp_children_list_str = "";
                for(String temp_child : temp_children_list){
                    if (temp_children_list_str == "") temp_children_list_str =
temp_child;
                    else temp_children_list_str += String.format(";%s", temp_child);
                }
```

```
                children_list.add(temp_children_list_str);
            }
            else if(child.getClass() == abap_newParser.St_ifContext.class)
children_list.add(visitSt_if((abap_newParser.St_ifContext) child_ctx).toString());
            else if(child.getClass() == abap_newParser.St_else_ifContext.class)
children_list.add(visitSt_else_if((abap_newParser.St_else_ifContext)
child_ctx).toString());
            else if(child.getClass() == abap_newParser.St_elseContext.class)
children_list.add(visitSt_else((abap_newParser.St_elseContext)
child_ctx).toString());
            else if(child.getClass() == abap_newParser.ConditionContext.class)
children_list.add((visitCondition((abap_newParser.ConditionContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Loop_assignContext.class)
children_list.add((visitLoop_assign((abap_newParser.Loop_assignContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Loop_intoContext.class)
children_list.add((visitLoop_into((abap_newParser.Loop_intoContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.AssignmentContext.class)
children_list.add((visitAssignment((abap_newParser.AssignmentContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.ValueContext.class)
children_list.add(child.getText());
            else if(child.getClass() == abap_newParser.St_caseContext.class)
children_list.add((visitSt_case((abap_newParser.St_caseContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.St_case_whenContext.class)
children_list.add((visitSt_case_when((abap_newParser.St_case_whenContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.KeywordContext.class)
children_list.add(child.getText());
            else if(child.getClass() == abap_newParser.OperatorContext.class)
children_list.add(child.getText());
            else if(child.getClass() == abap_newParser.OperandContext.class)
children_list.add(child.getText());
            else if(child.getClass() == abap_newParser.GenericContext.class)
children_list.add((visitGeneric((abap_newParser.GenericContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Method_callContext.class)
children_list.add((visitMethod_call((abap_newParser.Method_callContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Read_assignContext.class)
children_list.add((visitRead_assign((abap_newParser.Read_assignContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Select_intoContext.class)
children_list.add((visitSelect_into((abap_newParser.Select_intoContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Select_paramContext.class)
children_list.add((visitSelect_param((abap_newParser.Select_paramContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Where_paramContext.class)
children_list.add((visitWhere_param((abap_newParser.Where_paramContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Where_opContext.class)
children_list.add((visitWhere_op((abap_newParser.Where_opContext)
child_ctx).toString()));
            else if(child.getClass() == abap_newParser.Select_singleContext.class)
children_list.add((visitSelect_single((abap_newParser.Select_singleContext)
child_ctx).toString()));
        }
      }
      return children_list;
    }
}
```