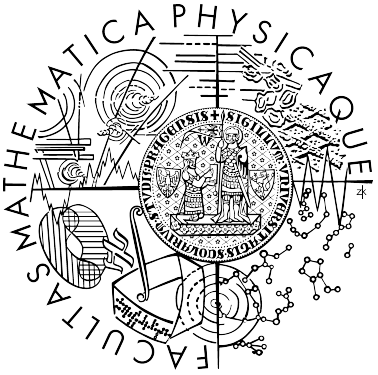
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Luděk Hlaváček

Makroprocesor

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.

Studijní program: Informatika, programování

2006

Na tomto místě bych rád poděkoval vedoucímu své práce RNDr. Tomáši Holanovi, Ph.D. za odborné vedení mé práce, za rady a za čas, který mi věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Jičíně dne 7.8.2006 Luděk Hlaváček

Obsah

Kapitola 1. Úvod **6**

1.1 Cíl 6
1.2 Obsah práce 6

Kapitola 2. Makroprocesory **7**

2.1 Co je to makroprocesor? 7
2.2 Makra 7
2.3 Parametry maker 7
2.4 Historie 8
2.5 Alternativy 9

Kapitola 3. Některé významnější makroprocesory **10**

3.1 GPM 10
3.2 ML/I 10
3.3 M4 11
3.4 gema 11
3.5 Preprocesor jazyka C 12
3.6 Java+ 12
3.7 SMX 12
3.8 TeX 13

Kapitola 4. Implementace **14**

4.1 Dokumentace 14
4.2 Výběr vývojového prostředí 14
4.3 Jádro makroprocesoru 15
4.4 Implementace vstupně-výstupních funkcí 15
4.5 Nevýhody existujících makroprocesorů 15
4.6 Schopnosti makroprocesoru PMP 15
4.7 Makra 16
4.8 Volání maker 16
4.9 Kódování výstupu volaných maker 17

Kapitola 5. Uživatelský manuál **18**

5.1 Instalace 18
5.2 ANT 18
5.3 GUI 18
5.4 Syntaxe vstupního textu 19
5.5 Uživatelsky definovaná makra 20

5.6 Vestavěná makra....20
5.7 Přidávání vestavěných maker....20
5.8 Historie definic makra....20
5.9 Další manipulace s makry....21
5.10 Podmíněné vyhodnocování....21
5.11 Přesměrování vstupu a výstupu....21
5.12 Ukázkové příklady....21

Kapitola 6. Závěr....22

6.1 Možnosti dalšího vývoje aplikace....22

LITERATURA....23

PŘÍLOHY....24

Přehled direktiv makroprocesoru....24
Vestavěná makra....26
Přehled speciálních symbolů pro definice uživatelských maker....31
Gramatika vstupního souboru....32
Konfigurační soubory....33
Parametry příkazové řádky....34
Ant Task....35
Obsah CD....37

Název práce: Makroprocesor

Autor: Luděk Hlaváček

Katedra (ústav): Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.

e-mail vedoucího: Tomas.Holan@mff.cuni.cz

Abstrakt: Cílem práce je navrhnout a implementovat univerzální makroprocesor. Vytvořený makroprocesor dovoluje používat běžné funkce, jako je podmíněné vyhodnocování, vkládání souborů, definice uživatelských maker a manipulace s makry. Dále je možné měnit nastavení makroprocesoru za běhu pomocí vestavěných příkazů a předefinovat vestavěné příkazy. Součástí je i několik demonstračních příkladů.

Práce obsahuje také srovnání s existujícími makroprocesory a stručně popisuje vývoj v této oblasti. Popsány jsou také obecné principy zpracování maker.

Klíčová slova: makroprocesory, zpracování textů, programovací jazyky, preprocessing

Title: Macro processor

Author: Luděk Hlaváček

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Tomáš Holan, Ph.D.

Supervisor's e-mail address: Tomas.Holan@mff.cuni.cz

Abstract: The goal of this work is to design and implement general-purpose macro processor. This macro processor supports common features such as conditional evaluation, file inclusion, user-defined macros and manipulations with macros in run-time. Various modifications of the configuration of the macro processor are possible in run-time as well. Also it is possible to change the way of invoking built-in commands Several examples are included to demonstrate implemented features.

The work also contains brief description, history and comparison with existing macro processors and theoretical principles of macro processing.

Keywords: macro processors, text processing, programming languages, preprocessing

Kapitola 1. Úvod

1.1 Cíl

Cílem této práce je vytvořit univerzální makroprocesor, který by měl být dostatečně flexibilní, aby mohl být použit pro všechny úlohy běžně prováděné makroprocesory. To znamená, že by měl umět volat makra s parametry, vkládat soubory, definovat a měnit makra za běhu, umožňovat podmíněné rozvíjení. Navíc by mělo být možné měnit nastavení makroprocesoru pomocí příkazů ve zdrojovém textu.

Vyvíjený makroprocesor dostal pracovní název *PMP*, který mu i zůstal.

1.2 Obsah práce

Práce je rozdělena do několika kapitol a příloh.

Druhá kapitola popisuje historii a vývoj makroprocesorů.

Třetí kapitola popisuje a rozebírá vlastnosti některých rozšířenějších makroprocesorů.

Čtvrtá kapitola popisuje vývoj a vlastnosti makroprocesoru vytvořeného v rámci této práce. Zaměřuje se na charakteristické vlastnosti ve srovnání s jinými makroprocesory.

Pátá kapitola popisuje práci s programem.

Závěrečná kapitola shrnuje dosažené výsledky a rozebírá možné směry dalšího vývoje programu.

Přílohy obsahují přehled vestavěných maker, seznam konfiguračních direktiv popis speciálních symbolů pro definovaná makra a gramatiku, podle které makroprocesor pracuje. Na závěr uveden obsah přiloženého CD

Kapitola 2. Makroprocesory

2.1 Co je to makroprocesor?

Makroprocesor je obecně program na zpracování textu, který v nejjednodušší formě pouze kopíruje vstupní text na výstup a přitom provádí předdefinované úpravy textu. Tato vágní definice naznačuje, že množina nástrojů spadajících do této kategorie nebude příliš jasně ohraničena.

2.2 Makra

Hlavní činností všech makroprocesorů je zpracovávání maker ve zdrojovém textu. Obvyklým přístupem, kterého se drží prakticky všechny makroprocesory, je vyhledávání nějakého speciálního identifikátoru nebo předem definovaného vzoru. Často to bývá název makra, který je uvozen symbolem, který určuje, že se jedná o volání makra. Tento princip demonstruje schéma 2.1.

Schéma 2.1: Princip práce makroprocesoru

Makra se dají dělit podle různých kritérií, která se navíc mohou lišit podle toho, co ještě za makra považujeme a co už ne. Často se makra rozdělují na dvě skupiny – na vestavěné příkazy a uživatelsky definovaná makra [13]. Většina makroprocesorů obsahuje sadu vestavěných příkazů, které slouží k provádění činností jako je například definování uživatelských maker nebo podmíněné vyhodnocování. Uživatelsky definovaná makra jsou určena svým substitučním řetězcem, kterým je nahrazeno jejich volání. Součástí tohoto řetězce mohou být symboly, které jsou při vyvolání makra nahrazeny obsahem daného parametru.

Jednotlivé makroprocesory se ale liší v tom, jak s těmito dvěma skupinami maker pracují. Jedním extrémem je například preprocesor jazyka C, který vyžaduje, aby příkazy pro makroprocesor byly uvozeny znakem '#', ale definovaná makra nijak uvozena nejsou a můžou mít za název libovolný platný identifikátor jazyka C. Na druhou stranu m4 a další univerzální makroprocesory se snaží o sjednocení přístupu k makrům, tj. co nejméně rozlišovat volání příkazů pro makroprocesor a uživatelsky definovaných maker.

2.3 Parametry maker

Makroprocesor také musí určit, nebo umožnit uživateli definovat způsob předávání parametrů volaným makrům [15]. Pokud by nebylo možné parametry makrům předávat, značně by to degradovalo schopnosti daného makroprocesoru.

Obvyklý způsob je definování symbolů uzavírajících parametry (*delimiters*) a symbolů pro oddělení jednotlivých parametrů (*separators*). Jednotlivé makroprocesory se liší hlavně ve způsobu zacházení s těmito symboly. Některé makroprocesory umožňují definovat tyto symboly zvlášť pro každé makro (*ML/I*, *gema*), u jiných jsou společné pro všechna makra (*m4*).

Vlastní způsob zpracování parametrů makra představuje další charakteristický rys makroprocesorů.

Nejstarší (a z hlediska implementace nejjednodušší) způsob je danému makru předávat parametry jako prostý text bez jakékoliv transformace. Tento postup vyžaduje pouze vyhledání oddělovačů a ukončovacího symbolu ve zdrojovém textu.

Pokud má makroprocesor parametry zpracovávat, nabízí se více možností, jak to udělat. Jednodušší postup je zpracovat přečtené parametry samostatně, jako by to byly samy o sobě nezávislé vstupní texty makroprocesoru. Obecnější metodou je rozšířit lexikální analýzu o symboly pro oddělování parametrů a ukončovací symbol a pokračovat ve čtení vstupu. Po přečtení ukončovacího symbolu pak dojde k vyvolání makra s přečtenými parametry.

Zde se nabízí otázka, kdy má dojít k přečtení definice volaného makra. Součástí parametrů by totiž mohlo být i volání makra, které mění definici makra, jehož parametry nyní zpracováváme. Častějším řešením je čtení definice před zpracováním parametrů. Protipříkladem je makroprocesor *GPM*, který to dělá v opačném pořadí.

2.4 Historie

První makroprocesory se objevily s nástupem assemblerů. Psaní programů jen pomocí instrukcí daného procesoru bylo (a pořád je) náročné a výsledné programy jsou značně nepřehledné. Proto vznikly první makroprocesory (**preprocesory**), které předzpracovaly zdrojový kód, obohacený o složitější jazykové konstrukce, do podoby, kterou dokázal přeložit assembler. Mezi běžné schopnosti těchto preprocesorů patří vkládání souborů, definice konstant a samozřejmě definice maker. Postupem doby se preprocesory staly nedílnou součástí assemblerů a zpracování maker se provádí obvykle přímo v assembleru.

Po assemblerech se začaly rozšiřovat vyšší programovací jazyky, u kterých nehrály makroprocesory tak významnou roli. Typickým příkladem je preprocesor jazyka C. Jeho funkce se prakticky redukovaly na definici maker a vkládání souborů.

Alternativou k preprocesorům představují univerzální makroprocesory. Ty obvykle nejsou vázány k nějakému konkrétnímu programovacímu jazyku, ale definují vlastní strukturu vstupního textu, nebo umožňují uživateli tuto strukturu definovat.

Další použití našly makroprocesory v oblasti generování a sazby textů. Většina systémů pro sazbu textu dovoluje uživateli nějakým způsobem používat a definovat makra. Nejznámějším příkladem je bezesporu *TeX*, ale makra podporuje i systém *groff*, (resp. jeho předchůdci *troff* a *nroff*), který je běžnou součástí unixových systémů.

Podobné uplatnění našly makroprocesory i v oblasti generátorů dokumentací a webových stránek. Často je zbytečné pro vygenerování stránky používat skriptovací jazyky, které provádějí syntaktickou analýzu celého dokumentu a výsledný program provedou. Proto se k tomuto účelu často používají univerzální makroprocesory. Existuje například

několik různých balíků maker pro makroprocesor m4 určených k vytváření HTML dokumentů [11].

Makra najdeme i v mnohých textových editorech, i když často se pod názvem makro skrývá nahrávání činnosti uživatele za účelem pozdějšího zopakování. Takovéto editory prakticky vždy obsahují interpret nějakého skriptovacího jazyka namísto makroprocesoru.

2.5 Alternativy

Ne vždy je nasazení makroprocesoru vhodné. To platí především pro preprocessing v programovacích jazycích. Jazykové konstrukce makroprocesoru představují cizorodý element daného jazyka a navíc často značně snižují čitelnost programu (například makra v C umožňují vkládat do těla makra prakticky cokoliv). Je tendence rozšiřovat jazyky vyšší úrovně tak, aby se funkce makroprocesoru přesunula překladač.

Proto dnes převažuje nasazení univerzálních makroprocesorů v oblasti zpracování textu (například vytváření konfiguračních souborů, formátování a sazba textu). Ale i zde jsou možnosti jak se makroprocesorů vyhnout. Buď pomocí skriptovacího jazyka, který umožní procedurální procedurální zpracování, nebo pomocí nástrojů na zpracování textu, jako například *sed* nebo *awk*.

Kapitola 3. Některé významnější makroprocesory

3.1 GPM

GPM je jedním z nejstarších univerzálních makroprocesorů [4]. Měl jednoduchou pevně danou strukturu volání maker a pevně dané speciální symboly. Nicméně už uměl provádět expanze maker v parametrech volání jiného makra.

Volání maker bylo uvozeno znakem '#', parametry se oddělovaly čárkou a ukončeny byly středníkem. Formální parametry se v těle makra označovaly znakem '&', který následovala číslice určující pořadí. Volání makra bylo ukončeno znakem středník. K definování maker sloužilo vestavěné makro *DEF*. Syntaxe volání vestavěných a definovaných maker byly stejné. Tento makroprocesor už počítal s tím, že občas může být potřeba vložit do těla makra rezervované znaky. Proto bylo možné zadávat textové řetězce uzavřené znaky '<' a '>'.

Zajímavou vlastností tohoto makroprocesoru bylo čtení definice makra až po zpracování jeho parametrů, takže bylo možné definovat makro uvnitř jeho parametrů. Následující příklady tuto vlastnost demonstrují [13]:

```
#A,X,Y,#DEF,A,B;;
```

výstup:

```
B
```

Složitější příklady jsou už náročnější na porozumění:

```
#DEF,SUC,<#1,2,3,4,5,6,7,8,9,10,#DEF,1,<&>&1;;>;
#SUC,2;
```

výstup:

```
3
```

3.2 ML/I

ML/I je dalším historickým makroprocesorem. I když je označován jako univerzální, tak jeho hlavním úkolem bylo rozšíření syntaktických konstrukcí soudobých programovacích jazyků. Dovoloval definovat strukturu volání zvlášť pro každé makro. Volání makra se definovalo jako posloupnost lexikálních elementů, z nichž některé mohly být označeny jako parametry makra. Definování maker a podobné činnosti byly prováděny pomocí vestavěných maker.

Tento makroprocesor ještě neuměl vnořená volání maker (expanzi maker v parametrech makra).

Příklady možného volání maker [14]:

```
DO {arg1} TIMES {arg2} REPEAT
MOVE FROM {arg1} TO {arg2} ;
INTERCHANGE ( {arg1} , {arg2} ) {NL}
```

3.3 M4

Programovací jazyk Fortran byl značným skokem ve vývoji programovacích nástrojů. Oproti assembleru umožňuje používat složitější konstrukce (jako např if, switch, cykly, apod…). Nicméně jeho syntaxe je značně těžkopádná a programy nejsou příliš přehledné. Proto i pro tento jazyk postupně vzniklo několik preprocesorů. Jedním z nich je i Ratfor [3]. Ten je založen na makroprocesoru m4, který od doby vzniku Ratforu získal značně na popularitě a v současnosti je standardní součástí většiny unixových systémů, díky čemuž je zřejmě nejpoužívanějším univerzálním makroprocesorem. Mezi nejčastěji používané programy, které m4 využívají, patří *Autoconf*, který slouží ke generování skriptů pro kompilování programů na různých platformách variantách Unixu. Mezi další ukázky široké palety použití tohoto makroprocesoru patří programy *Bison* nebo *sendmail*.

M4 má oproti ostatním makroprocesorům relativně málo vestavěných maker (okolo 50). Částečně je to dáno vazbou na unixové operační systémy, které poskytují bohatou sadu utilit, které chybějící makra suplují.

Mimo zpracování maker umí m4 rozpoznávat i znakové řetězce a komentáře [5]. Znakové řetězce jsou ohraničeny počáteční a koncovou posloupností znaků. Řetězce je možné vnořovat, což znamená, že v každém řetězci musí být vyvážený počet počátečních a ukončovacích symbolů. Hodnotou řetězce je jeho obsah po odstranění vnějšího páru ohraničujících symbolů.

Komentáře jsou chápány jako část vstupního textu, která se kopíruje na výstup makroprocesoru bez jakýchkoliv úprav (neprovádí se expanze maker). Skutečné komentáře je možné vkládat pomocí vestavěného makra *dnl*, které při zpracování odstraní všechen text až do konce řádku.

Novější verze m4 umožňují změnit symboly pro uvození a ukončení řetězců i komentářů. K tomu slouží vestavěné příkazy *changequote* a *changecom*.

M4 nabízí široké možnosti pro manipulaci s makry. Běžný způsob představují příkazy *define* a *undefine*. Další možností jsou příkazy *pushdef* a *popdef*, které pracují se zásobníkem definic, který má každé makro přiřazeno. Příkaz *pushdef* uloží současnou definici makra na zásobník a nahradí ji zadaným výrazem. Naopak příkaz *popdef* současnou definici zruší a nahradí ji hodnotou načtenou z vrcholu zásobníku.

Dále je také možné manipulovat s výstupním textem. Pomocí makra *divert* lze všechen následující výstup přesměrovat do vyrovnávací paměti a pokud nebude smazán pomocí makra *cleardivert*, připojí se na konec výstupu, nebo na místo určené voláním makra *undivert*. Podobnou funkci má i makro *m4wrap*, které zadaný text připojí na konec vstupního textu [1, 2].

3.4 gema

Co se týká možností definování způsobu volání maker, je gema zdaleka nejpokročilejší makroprocesor. Celé volání makra je definováno regulárním výrazem (nazývaným *template*), ve kterém navíc lze určit umístění a možný obsah parametrů předávaných volanému makru. Vestavěné příkazy se volají pomocí identifikátorů zadaných v těle uživatelského makra (*action*).

Příklad, který vyhledá a vypíše texty `"Hello x!"` (kde *x* je libovolné slovo) spolu se jménem souboru a číslem řádku. Znakem `'@'` jsou uvozeny příkazy pro makroprocesor a ve složených závorkách `'{}'` jsou uzavřeny jejich parametry.

```
"Hello *\!=@err{@file řádek @line\: Nalezeno Hello $1}"
```

Toto je jen velmi jednoduchý příklad. Díky velkému množství parametrů a přepínačů lze vytvářet značně složité definice maker. To umožňuje zpracovávat i vstupy se složitou lexikální a syntaktickou strukturou [7].

3.5 Preprocesor jazyka C

Preprocesor je pevnou součástí jazyka C [6]. Nejčastěji používaná makra jsou makra *#include* (vložení souboru), makra pro podmíněný překlad (*#if*, *#ifdef*, *#else*, …).

Často používaná je i makro *#define*, kterým se definují uživatelská parametrická i bezparametrická makra. Ta často slouží k zajištění přenositelnosti programů. V knihovnách se pomocí maker oddělí platformě nezávislé rozhraní od platformě závislých datových typů a deklarací. Nicméně už ani jazyk C není na tomto přímo závislý. Součástí jazyka jsou konstrukce, které makra přímo nahradí (*const*, *typedef*, *inline* funkce).

Další makra *#line*, *#error*, *#pragma* jsou jen řešením, jak do programu vkládat metadata. Novější jazyky potvrzují trend odklonu od makroprocesorů a umožňují vkládání metadat přímo v syntaxi jazyka. Příkladem jsou anotace v Javě a v jazycích pro *.NET*.

3.6 Java+

Java+ je jedním z několika nástrojů pro Javu, který by se dal ještě nazvat makroprocesorem. Zatím nemá zabudovánu přímo podporu pro zpracování maker, ale umožňuje programátorovi zadávat řetězce jednodušším a přehlednějším způsobem než je tomu v Javě.

Java+ je hlavně ukázkou klesajícího významu makroprocesorů v moderních programovacích jazycích (obzvláště v těch objektově orientovaných).

Použití demonstruje následující příklad [8]:

Java+

```
System.out.println({{
<xmlExample>
<name>{{myName}}</name>
<number>{{myNumber}}</number>
</xmlExample>}});
```

Java

```
System.out.println("\n"+
"<xmlExample>\n"+
"<name>"+myName+"</name>\n"+
"<number>"+myNumber+"</number>\n"+
"</xmlExample>");
```

3.7 SMX

SMX je příkladem makroprocesoru určeného ke generování dokumentů. Původně byl součástí web serveru Internet Factory's Commerce Builder [16]. V současnosti existuje několik implementací včetně modulu pro server Apache.

Hlavními výhodami tohoto makroprocesoru je jednoduchá syntaxe a množství vestavěných maker.

Zajímavý je způsob redefinice existujících maker. Podobně jako m4 má i SMX pro každé makro zásobník definic, ale lze do něj definice pouze přidávat. Vyvolání předchozí definice, se provádí přidáním požadovaného množství znaků ':' před název makra. Znak '/' vyvolá nejstarší definici makra.

Řetězce se zadávají v uvozovkách. Druhou možností, jak zabránit nechtěné expanzi maker v parametrech, je znak apostrof.

Jednoduchý příklad zpracování formuláře [16]:

```
%expand%
%if(%equal(%form(Name),'Joe),
    "Hello Joe!",
    'I don't know you!)
```

3.8 TeX

TeX je ukázkou sázecího programu s vestavěným makroprocesorem. Uživatel si může nadefinovat makra, která jsou expandována před vlastním typografickým zpracováním. TeX obsahuje několik set vestavěných příkazů (primitiv). Z větší části to jsou typografické příkazy, ale jsou mezi nimi samozřejmě i příkazy pro práci s makry a řídící příkazy (např. makro umožňující podmíněné vyhodnocení).

Kapitola 4. Implementace

4.1 Dokumentace

Prvotním dokumentem popisujícím vyvíjený makroprocesor byla **specifikace ročníkového projektu**. Jejím obsahem je návrh architektury makroprocesoru a uživatelského rozhraní. Od jejího vypracování už nedošlo k žádným podstatným změnám v architektuře. Byly jen upraveny a rozšířeny možnosti nastavení makroprocesoru a přidána podpora pro Ant.

Detailní uživatelská příručka obsahující popis instalace a používání makroprocesoru je součástí **dokumentace k ročníkovému projektu**. Navíc obsahuje obecnější programátorskou dokumentaci popisující základní rozhraní a strukturu makroprocesoru.

Detailní informace o implementaci lze nalézt v automaticky generované dokumentaci **Javadoc**.

Použití makroprocesoru v rámci nástroje Ant je popsáno v **manuálu k úloze PMPTask**.

Všechny tyto dokumenty jsou přiloženy na doprovodném CD ve složce *dokumenty*.

4.2 Výběr vývojového prostředí

Už od začátku byl PMP koncipován jako makroprocesor pro Javu. Původně byl psán pro Javu 1.4. S nástupem Javy 1.5 a s přihlédnutím k jejím novým funkcím došlo k přechodu na novou verzi. Některé nové vlastnosti, jako například parametrizované kolekce (*generics*), výčtové typy (*enums*) a statické importy, pomohly zpřehlednit kód a odstranit některé chyby.

Přechod k verzi 1.6 se sice neplánuje, ale bylo ověřeno, že makroprocesor bez problémů funguje i v této verzi.

Spolu s přechodem mezi verzemi Javy došlo i ke změně vývojového prostředí. Původní *Borland JBuilder 9* byl nahrazen prostředím *Eclipse 3.1*, které bez problémů funguje i Javě 1.5.

Jednou z hlavních výhod Javy je rozsáhlá knihovna, která je součástí prostředí JRE (*Java Runtime Environment*). I přesto se ukázalo, že napsání celého makroprocesoru s dostatečně funkčním rozhraním, které by poskytovalo i nadstandardní funkce jako například dávkové zpracování vstupních souborů, jen s využitím systémových knihoven by bylo značně náročné a neúčelné, protože .jsou k dispozici efektivní a vyzkoušená řešení. Proto bylo vytvořeno rozhraní umožňující spouštět PMP pomocí nástroje Ant.

Ant umožňuje automatizovat prakticky všechny činnosti spojené s vývojem programů v Javě. Sice není součástí Javy, ale většina vývojových prostředí (např Eclipse, NetBeans, JBuilder) ho má zabudovaný nebo s ním aspoň umí pracovat. Vzhledem k možnostem, které Ant nabízí, se ukázala integrace s tímto nástrojem jako velmi přínosná. Ant poskytuje úlohám, které spouští, řadu funkcí, které by bylo jinak nutné implementovat pro každou úlohu zvlášť. Mezi hlavní funkce, které PMP využívá, patří dávkové zpracování více souborů (pomocí úlohy *FileSet*), transformace jmen vstupních

souborů na jména výstupních souborů (úloha *Mapper*) a načítání Java knihoven (úloha *TypeDef*).

4.3 Jádro makroprocesoru

Lexikální analýza je tou částí projektu, u které byl kladen největší důraz na flexibilitu a konfigurovatelnost. Výsledkem je možnost zadat pomocí regulárních výrazů formát každého lexikálního elementu [9]. V porovnání s makroprocesorem *gema* to sice nedovoluje takovou volnost při specifikaci formátu volání maker, ale nastavení je jednodušší a přehlednější. Celková koncepce zpracování vychází z makroprocesoru m4, který ale zatím nemá tak velké možnosti nastavení lexikální analýzy, nebo jsou zatím jen v experimentální fázi.

Jednou z možností implementace bylo použít jeden z dostupných generátorů lexikálních analyzátorů. Například JLex, nebo JFlex. To se ale ukázalo jako značně problematické, vzhledem k požadavku na změny lexikální analýzy za běhu makroprocesoru.

Použitelnější by byl generátor syntaktického analyzátoru, např. JavaCC, ANTLR, CUP, nebo BYACC/J. Nakonec se ale ukázalo, že syntaktická analýza není dostatečně složitá, aby se nasazení externího programu vyplatilo. Implementována je tedy zásobníkovým automatem, který zajišťuje načítání lexikálních elementů podle dané gramatiky a úkoly spojené s vnořenými voláními maker.

4.4 Implementace vstupně-výstupních funkcí

Aby byl makroprocesor užitelný v různých prostředích, jsou vstupně výstupní funkce reprezentovány jednou třídou, která zajišťuje vytváření standardních objektů *Reader* a *Writer*. Součástí standardní instalace je implementace pracující nad soubory a nad objekty type *String* (používá se v grafickém rozhraní). Uvnitř makroprocesoru je pak fronta, která se stará o postupná čtení ze vstupních objektů, což umožňuje jednoduchou implementaci makra *include*.

4.5 Nevýhody existujících makroprocesorů

Nevýhodou současných makroprocesorů je nízká míra modifikovatelnosti a rozšiřitelnosti. Makroprocesor obvykle má pevně danou množinu příkazů (vestavěných maker), kterou nelze nijak rozšířit. Pokud je potřeba, aby makroprocesor udělal něco, co nelze provést pomocí vestavěných maker, obvykle se nějakým způsobem zavolá externí program. To je ale značně pomalé a hlavně to komplikuje přenositelnost programu a tedy snižuje i vlastní přínos daného makroprocesoru.

PMP umožňuje psát jednoduchým způsobem nová makra, nebo vytvářet skripty interpretované makroprocesorem pomocí maker v balíku *pmp.macro.java*.

4.6 Schopnosti makroprocesoru PMP

Vyvíjený makroprocesor vychází převážně z makroprocesoru m4. Ale díky tomu, že je implementován v Javě, může využívat možností, které nabízí Java nabízí. Mezi ně

patří načítání modulů za běhu, podpora různých znakových sad a podpora regulárních výrazů přímo v systémových knihovnách.

Stejně jako m4 má sadu vestavěných maker, která lze použít k definování maker uživatelských. Dále dokáže pracovat s různými typy řetězců, které mají značně flexibilní možnosti definování.

4.7 Makra

Podobně jako jiné makroprocesory, umožňuje i PMP zadávat definice maker na příkazovém řádku. Pro nestandardní nastavení lexikální analýzy je ale potřeba definovat větší množství maker, u kterých by navíc nebylo jednoduché (a hlavně přenositelné) zabránit interpretaci speciálních znaků příkazovým interpretrem. Proto je možné zadat definice maker v konfiguračním souboru, který má formát XML.

Nastavení makroprocesoru se provádí pomocí **konfiguračních direktiv.** Aby byla zachována jednoduchost a konzistentnost celého makroprocesoru, jsou direktivy součástí tabulky maker. Až na několik málo výjimek jsou reprezentovány uživatelsky definovanými makry. Konfliktům s používanými makry je zabráněno pomocí společné předpony, která navíc zabraňuje nechtěnému vyvolání. Ve výchozím stavu je tato předpona **"**`pmp`**:"**.

Sloučení maker a konfiguračních direktiv umožnilo vyhnout se dvěma sadám různých maker, elementů v konfiguračním souboru a parametrů příkazové řádky.

Aby mohl makroprocesor okamžitě reagovat na změny konfiguračních direktiv, je mezi komponentami implementován mechanismus *event*/*listener* ze standardních knihoven Javy [12].

Uvnitř makroprocesoru je každé makro reprezentováno jednou třídou (*Java class*). Uživatelsky definovaná makra jsou instance třídy *pmp.DefinedMacro*, ostatní makra jsou instancemi nějakého jiného potomka společné třídy všech maker *pmp.AbstractMacro.* Makroprocesor s makry manipuluje pomocí tabulky maker, což komponenta založená na hašovací tabulce *java.util.Hashtable*, která umožňuje rychlé vyhledání makra podle jeho jména. Zde trochu komplikuje situaci možnost makroprocesoru ignorovat velikost písmen ve volání maker (direktiva. *macro.casesensitive*, viz přílohy). To by vyřešila další tabulka maker, kde by byla makra indexována názvy převedenými na malá písmena. Nicméně se ukázalo, že zpomalení není příliš výrazné (díky malé velikosti tabulky) a ani sama tato direktiva nebude pravděpodobně příliš používaná.

4.8 Volání maker

Následující postup se provede při nalezení lexikálního elementu, který odpovídá volání makra.

1. z přečteného tokenu se zjistí jméno volaného makra
2. zjistí se, zda dané makro existuje (hledání se případně rozšíří podle nastavení direktiv *macro.casesensitive* a *macro.fallback*
3. pokud dané makro neexistuje, vypíše se přečtený token na výstup a pokračuje se krokem 8

4. pokud není následující token symbol pro uvození parametrů (dle direktivy *macro.opening.bracket*), pak je množina parametrů prázdná a pokračuje se krokem 6 (pokud je nastavena direktiva *macro.args.required*, pak se vypíše přečtený token a pokračuje se na 8)

5. čtou se parametry makra dokud se nenarazí na token odpovídající direktivě *macro.closing.bracket*, přičemž při každém tokenu *macro.paramseparator* se začne ukládat do zásobníku nový parametr

6. zavolá se makro s danými parametry

7. výsledek volání makra se vloží do vstupního textu za aktuální pozici

8. zpracovávání vstupního textu pokračuje dál

4.9 Kódování výstupu volaných maker

S voláním maker úzce souvisí jeden problém. Výsledkem expanze makra může být posloupnost znaků, kterou makroprocesor nějak interpretuje. Například jako další volání makra. Obecně nelze zajistit, aby výstup makra neobsahoval speciální symboly (příkladem může být volání makra Exec). Proto je nutné poskytnout mechanismus, který dalšímu zpracování výstupu maker zabrání.

Makroprocesor PMP pro tento účel poskytuje funkci kódování výstupu, která umožňuje zpracovávat výstup daného makra jiným makrem. Typickým příkladem, který dobře funguje ve výchozím nastavení, je makro *pmp.macro.lex.CEncoder*. To z textu na svém vstupu udělá řetězec znaků podle pravidel jazyka C.

Tuto funkci je možné používat jak u vestavěných maker (jako parametr makra *builtin*), tak pro uvození parametrů v uživatelsky definovaných makrech.

Kapitola 5. Uživatelský manuál

5.1 Instalace

Celý makroprocesor je umístěn ve spustitelné Java knihovně pojmenované *pmp.jar*. Spuštění makroprocesoru se provede následujícím příkazem:

```
java -jar pmp.jar
```

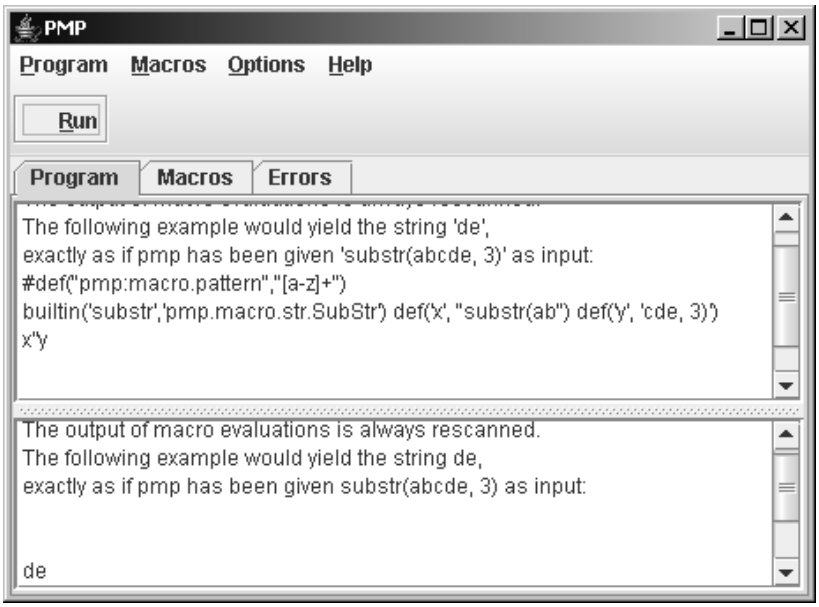
5.2 ANT

Návod jak používat Ant je součástí manuálu [10].

Pro použití makroprocesoru v prostředí nástroje Ant je potřeba buď zkopírovat knihovnu *pmp.jar* do adresáře *$ANT_HOME/lib*, nebo ji importovat za běhu pomocí úlohy *TaskDef*. Parametry pro spouštění makroprocesor u jsou popsány v manuálu k úloze PMPTask (viz příloha).

5.3 GUI

Grafické rozhraní slouží hlavně k otestování funkčnosti složitějších maker.



Obrázek 4.1 Hlavní okno grafického rozhraní

Nabídka program obsahuje příkaz *Run* pro spuštění zadaného programu a příkaz *Quit*, který ukončí grafické rozhraní.

Nabídka **Macros** obsahuje příkazu pro manipulaci s makry. Položka *Load default configuration* provede znovunačtení výchozího nastavení makroprocesoru. K načtení konfigurace z externího souboru slouží příkaz *Load config file*. Přepínač *Always reload macros* zajistí obnovení výchozího stavu před dalším spuštěním programu.

Nabídka **Options** obsahuje pouze volbu *Show error dialog*, která umožňuje zakázat zobrazování chybového dialogu při výskytu chyby. Bez ohledu na stav této volby, jsou všechny chyby zaznamenávány do seznamu v záložce *Errors*.

Panel nástrojů obsahuje zatím pouze tlačítko pro spuštění programu (*Run*).

Hlavní částí okna jsou záložky *Program*, *Macros*, *Errors*. Záložka **Program** je rozdělena na dvě části. Editační box v horní polovině slouží k zadání vstupního programu. V dolní části se zobrazí výstup makroprocesoru. Záložka **Macros** obsahuje přehled maker. Tlačítko *Change* slouží ke změně definovaného makra. Pokud není zaškrtnuta volba *Always reload macros*, bude při příštím spuštění makroprocesoru použita tato změněná hodnota. Pod záložkou **Errors** je umístěn seznam chybových a varovných hlášení.

5.4 Syntaxe vstupního textu

Znakové řetězce

Na rozdíl od jiných makroprocesorů nemá PMP pevně daný formát znakových řetězců a komentářů. Oba typy literálů se definují stejným způsobem a je možné současně definovat libovolné množství formátů. Každý typ řetězce (resp. komentáře) je určen svým identifikátorem, díky kterému je možné definovat jeho vlastnosti nezávisle na ostatních typech řetězců.

Direktivy pro nastavení vlastností řetězce mají tvar string.*identifikátor*.*vlastnost.* Pro nastavení symbolů uvozujících a ukončujících řetězec slouží direktivy *string.*.start*, *string.*.end*. Stejně jako v případě jiných lexikálních elementů, je i zde očekáván regulární výraz.

Způsob zpracování řetězce určuje direktiva *string.*.mode*. Hodnota *discard* odpovídá komentářům. Ostatní hodnoty reprezentují různé způsoby zpracování řetězcových literálů. Další direktivy ovlivňující formát literálu jsou popsány v příloze.

Volání maker

Volání maker má jednu z následujících forem

```
macroname
```

což je volání makra bez parametrů, nebo

```
macroname(paremeter1, parameter2, ..., parametern)
```

což je volání makra s parametry 1 až n. Makra můžou mít libovolný počet parametrů. Vestavěná makra dodržují konvenci, že přebytečné parametry se ignorují a chybějící nahrazují prázdným řetězcem. U uživatelsky definovaných maker se vygeneruje varování, pokud je v substitučním řetězci odkaz na nezadaný parametr.

Formát identifikátoru makra je dán direktivou *macro.pattern*. Ve výchozím nastavení musí být název makra uvozen znakem '#', jinak k expanzi makra nedojde.

Další direktivou, která ovlivňuje volání maker je *macro.args.required*, která povoluje, resp. zakazuje volání maker bez parametrů. Zda se mají rozlišovat malá a velká písmena určuje direktiva *macro.casesensitive*. Pokud není makro s daným jménem nale-

zeno, zkusí ještě makroprocesor direktivu *macro.fallback*. Pokud ani ta není definována, celé volání se zruší a část vstupního textu odpovídající danému volání se pošle na výstup. To také znamená, že pokud se volání neprovede, následující parametry se budou zpracovávat dále, jako by nebyly součástí tohoto neúspěšného volání.

Formát závorek a oddělovačů definují pomocí regulárních výrazů makra *macro.opening.bracket*, *macro.closing.bracket*, *macro.paramseparator*.

Další možností jak zabránit nechtěnému rozvíjení maker a zpracování řetězců, je rozdělení vstupního textu na sekce s čistým textem a na bloky s makry. Pomocí direktiv *code.start* a *code.end* se určí symboly, které uzavírají sekce, kde se budou zpracovávat makra. Zbytek vstupního textu se pouze kopíruje na výstup.

5.5 Uživatelsky definovaná makra

Stejně jako v jiných makroprocesorech, může v PMP uživatel definovat textová makra. Ta jsou určena pouze svým expanzním řetězcem, kterým je nahrazeno volání dotyčného makra ve zdrojovém textu. K manipulaci s těmito makry slouží vestavěná makra *define*[1] a *undefine*.

Aby bylo možné používat v definovaných makrech parametry, jsou definovány speciální symboly, které jsou při expanzi makra nahrazeny parametry volaného makra (viz příloha).

5.6 Vestavěná makra

Jako vestavěná (*built-in*, nebo jen *builtin*) makra se označují vnitřní příkazy makroprocesoru. V případě PMP je to trochu zavádějící název, protože uživatel může sadu těchto příkazů snadno rozšířit o svoje vlastní.

5.7 Přidávání vestavěných maker

Jediným rozdílem mezi vestavěnými makry dodávanými s makroprocesorem a těmi, která vytvořil uživatel, je jejich umístění. Makra dodávaná s makroprocesorem jsou umístěna v archivu spolu s ostatními částmi makroprocesoru, takže jsou pro Javu viditelné. U maker přidaných uživatelem je nutné interpretru sdělit jejich umístění. K tomuto účelu je určen element konfiguračního souboru *library*. Ant má pro tento účel úlohu *TaskDef* s parametrem *classpath* [10].

Vlastní mačtení se provede v konfiguračním souboru (element *class*), nebo v programu pomocí makro *builtin*.

Ve vstupním programu se pak už používání těchto maker nijak neliší.

5.8 Historie definic makra

Stejně jako například v makroprocesoru m4, může i zde uživatel pracovat s identifikátorem makra jako se zásobníkem definic. Pokud místo makra *define* budeme

1 Dále v textu je použita konvence, že pokud má vestavěné makro přiřazen ve výchozím nastavení název, je uveden tento název. Jinak je uveden název třídy.

používat *pushdef*, dojde k aktivaci zásobníku, na který se budou ukládat předchozí definice a ze kterého se následně budou obnovovat při volání makra *popdef*.

5.9 Další manipulace s makry

K vypsání definice uživatelsky definovaného makra slouží vestavěné makro *pmp.macro.Defn*. Makra *pmp.macro.IsBuilti*n a *pmp.macro.IsSet* slouží ke zjištění dalších informací o daném makru.

Pro nepřímé volání maker se používá *pmp.macro.Call*. To umožňuje volat i makra, která nelze volat přímo, protože nemají název odpovídající regulárnímu výrazu v direktivě *macro.pattern*.

Export všech maker nebo import maker zajišťuje makro *macros*. Toto makro používá stejný XML formát jako vlastní makroprocesor (viz příloha).

5.10 Podmíněné vyhodnocování

K podmíněnému vyhodnocování slouží makro *ifelse.* V závislosti na neprázdnosti prvního parametru se vrátí obsah druhého, resp. třetího parametru. Alternativou je makro *pmp.macro.math.IfNotZero*, které na místě prvního parametru očekává aritmetický výraz.

5.11 Přesměrování vstupu a výstupu

Součástí makroprocesoru jsou vestavěná makra *Include* a O*utput*. Volání makra *Include* se nahradí obsahem zadaného souboru. Makro *Output* přesměruje výstup makroprocesoru.

5.12 Ukázkové příklady

V adresáři *samples* je několik souborů, na kterých jsou demonstrovány různé aspekty práce makroprocesoru.

V souboru *README.txt* je vždy stručný návod k použití.

Kapitola 6. Závěr

V rámci této práce byl vytvořen univerzální makroprocesor se všemi požadovanými vlastnostmi. Důraz byl kladen hlavně na široké možnosti nastavení a možnosti změn nastavení za chodu makroprocesoru. To se odrazilo v rychlosti práce, která je v porovnání s jinými nástroji relativně nízká.

Nicméně poskytuje dostatek funkcí, díky kterým by mohl najít uplatnění i v praxi.

6.1 Možnosti dalšího vývoje aplikace

Lexikální analýza současné verze makroprocesoru je postavená na kolekci objektů *java.util.regex.Matcher*, což přináší několik různých problémů. Z hlediska rychlosti je nepříjemné, že pro každý typ lexikálního elementu, který může být v daný okamžik vydán, se provede porovnání vzoru s aktuálním místem v programu. Ideálním řešením by bylo vytvoření jediného vyhledávacího automatu, tak jak to dělají různé generátory lexikálních analyzátorů. Vytvořit generátor automatů, které by porovnávali vstup s množinou regulárních výrazů. které by měli syntaxi kompatibilní s knihovnou *java.util.regex,* je mimo rozsah této práce.

Grafické rozhraní nemá příliš funkcí, protože je méně podstatnou částí celého projektu. Určitě je možné rozšířit možnosti manipulace s makry. Dalším podstatným vylepšením by bylo přidání režimu krokování s možností definovat breakpointy podobně jako to dělají běžné debuggery.

LITERATURA

[1] Kernighan B. W., Ritchie D. M.: *The M4 macro processor. Technical report*, Bell Laboratories, Murray Hill, New Jersey, USA, 1977.

[2] Turner K. J.: E*xploiting the m4 macro language,* Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, Scotland, September 1994.

[3] Kernighan B. W.: *RATFOR – A Preprocessor for a Rational Fortran*, http://www.mit.edu:8001/afs/athena.mit.edu/astaff/project/docsourc/doc/unix.manual.progsupp2/08.ratfor/ratfor.PS

[4] Stratchey C.: *A General Purpose Macro generator*, Computer Journal 8,3 (1965), str. 225-241

[5] *GNU M4 Manual,* GNU Press. 2004, http://www.gnu.org/software/m4/manual/

[6] Brodský J., Skočovský L.: *Operační systém UNIX a jazyk C*, SNTL Praha, 1989

[7] *GEMA - The general purpose macro processor*, http://gema.sourceforge.net/

[8] Cox B.: *Java+ Preprocessor Release 2.0*, http://virtualschool.edu/java+/

[9] Friedl J. E. F.: *Mastering regular expressions - 2nd ed.,* O'Reilly, 2002, http://regex.info/

[10] *Ant manual*, http://ant.apache.org/manual/index.html

[11] Hepple B.: *Using m4 to write HTML*, http://linuxgazette.net/issue22/using_m4.html

[12] *Java API documentation*, http://java.sun.com/docs/

[13] Gries D.: *Compiler construction for digital computers*, Wiley New York, 1971

[14] Eager R.D., Brown P.J.: *ML/I User's Manual,* http://www.ml1.org.uk

[15] Brown P.J.: *A survey of macro processors*, Communications of the ACM 10, Oct 1969

[16] *SMX: Server Macro Expansion*, http://www.smxlang.org/

PŘÍLOHY

Přehled direktiv makroprocesoru

source.append
Text přidaný na začátek vstupního textu. Připojení se provádí před začátkem zpracování programu, takže je nutné tuto direktivu nastavit z vnějšku. Nicméně je možné sem vložit volání makra, které bude definováno až v programu.

source.prepend
Text připojený na začátek programu. Připojení se samozřejmě provádí před začátkem zpracování programu, takže je nutné tuto direktivu nastavit z vnějšku.

code.start
Značka pro uvození kódu (bloku s makry). Pokud je nastaveno na prázdný řetězec, přejde se do kódu okamžitě.
Výchozí nastavení: prázdné

code.end
Značka pro ukončení kódu.
Výchozí nastavení: prázdné

code.marks.output
Určuje zda se mají vypisovat na výstup značky pro uvození a ukončení kódu.
Výchozí nastavení: makro není definováno

code.token
Formát lexikálních elementů v kódu.
Výchozí nastavení: `[A-Za-z0-9_]+`

macro.pattern
Regulární výraz definující možná jména makra. Je možné používat *capturing groups* (viz nápověda k *java.util.regex.Pattern*), jméno makra je pak získáno z jejich spojení v daném pořadí.
Výchozí nastavení: `#([a-z]+)`

macro.opening.bracket
Definuje lexikální symbol uvozující parametry makra.
Výchozí nastavení: `[ \t]*\([ \t]*`

macro.closing.bracket
Definuje lexikální symbol ukončující parametry makra.
Výchozí nastavení: `[ \t]*\)`

macro.paramseparator
Oddělovač parametrů. Slouží k oddělení parametrů volaných maker.
Výchozí nastavení: `[ \t]*\,[ \t]*`

macro.casesensitive
V názvech maker se nerozlišuje velikost písmen. Pokud je zapnuto, pak se při volání nerozlišuje velikost písmen. Tuto direktivu není vhodné používat, protože zpomaluje práci makroprocesoru.
Protože lze vždy vytvářet makra, jejichž název se liší pouze velikostí písmen, může danému volání odpovídat více maker. V tomto případě je použito makro, které přesně odpovídá volání a pokud není definováno, pak se náhodně použije některé jiné (není dáno které, ani není zaručeno, že to bude vždy to samé).
Výchozí nastavení: `true`

macro.args.required
Určuje zda za názvem makra musí následovat seznam parametrů. Pokud je nastaveno na `true`, pak je makro vyvoláno jen pokud za ním následují parametry (tj: další token odpovídá výrazu v *macro.opening.bracket*.
Výchozí nastavení: `false`

macro.fallback

Makro, které vyvoláno, pokud se ve zdrojovém textu najde token, který je identifikován jako volání makra, ale příslušné makro není definováno. Pokud toto makro není definováno, pak se volání makra neprovede a přečtený identifikátor je pouze vydán na výstup.
Výchozí nastavení: makro není definováno

prefix

Určuje prefix konfiguračních direktiv. Změna této direktivy se okamžitě promítne do všech části makroprocesoru, které s její hodnotou pracují.
Výchozí nastavení: `pmp:`

string.*.start

Značka pro uvození řetězce.

string.*.end

Značka pro ukončení řetězce. Pokud není nastaveno, použije se stejný symbol jako pro uvození řetězce.

string.*.marks-mode

Určuje co se má udělat se značkami pro uvození a ukončení řetězce.
copy – kopírují se na výstup
copy-start – uvozující značka se kopíruje na výstup
copy-end – ukončovací značka se kopíruje na výstup
discard – zahodí se

string.*.tokens

Definuje formát lexikálních elementů v řetězci.

string.*.symbols

Formát speciálních symbolů zpracovávaných parserem.

string.*.parser

Parser speciálních symbolů pro tento typ řetězce. Tomuto makru se jako jediný parametr předá nalezený symbol.

string.*.mode

Určuje způsob zpracování obsahu řetězce.
copy – obsah řetězce se kopíruje na výstup
discard – obsah řetězce se zahodí
parse – speciální symboly se se před vydáním zpracují přiřazeným parserem
nested – řetězec s vnořenými elementy

string.*.value

Určuje text, kterým se nahradí obsah řetězce, pokud je nastaven režim *discard*.

string.*.standalone

Určuje zda tento typ řetězce má být rozpoznáván i mimo parametry makra. Pokud zde není nastaveno `true`, pak tento řetězec bude rozpoznáván kdekoliv v kódu.

defined.encoder

Makro, které slouží z zakódování parametrů v definovaných makrech. Používá se k transformaci řetězců tak, aby v nich bylo možné používat znaky, které by se jinak přeložily jako speciální symboly.

defined.separator

Oddělovač parametrů v definovaných makrech.

Vestavěná makra

Vestavěná makra jsou definována jako třídy v balících (*packages*) odvozených od *pmp.macro* (např. *pmp.macro.Include*) a výchozí stav u většiny z nich je, že jsou definována v tabulce maker pod svým jménem bez názvu balíku.

V následujícím seznamu je u maker, která jsou použita ve výchozím nastavení, uvedeno v závorce přiřazené jméno.

Základní vestavěná makra

pmp.macro.Include (include)
Provede vložení zadaného souboru na vstup makroprocesoru.
Parametry: jméno souboru (URI)

pmp.macro.Output (output)
Provede přesměrování výstupu do jiného souboru. Je možné zadat speciální identifikátory *pmp://stdout*, *pmp://stderr* a *pmp://null*.
Parametry: jméno výstupního souboru (URI)

pmp.macro.Error (error)
Ukončí zpracování a vypíše chybu.
Parametry: text chybové zprávy

pmp.macro.Warning (warning)
Vypíše varování.
Parametry: text varovné zprávy

pmp.macro.Echo (echo)
Vypíše daný text přímo na výstup.
Parametry: text

pmp.macro.Define (def)
Definuje nové makro.
Parametry: jméno makra
obsah makra

pmp.macro.Builtin (builtin)
Načte třídu s makrem. Třída musí být potomkem *pmp.AbstractMacro*.
Parametry: jméno makra
jméno třídy
[kódovač výstupu]

pmp.macro.IsBuiltin
Vrací `"1"`, pokud je zadané makro vestavěné. Pokud je definované, pak vrací `""`.
Parametry: jméno makra

pmp.macro.IsDefined (ifdef)
Testuje, zda se jedná o uživatelsky definované makro. Vrací `"1"` pokud je makro definováno, jinak prázdný řetězec. Toto makro je jen komplement k *IsBuiltin*.
Parametry: jméno makra

pmp.macro.IsSet (ifdef)
Testuje, zda je makro definováno. Vrací `"1"` pokud je makro definováno, jinak prázdný řetězec.
Parametry: jméno makra

pmp.macro.Undefine (undef)
Zruší definici makra. Nefunguje pokud bylo makro vytvořeno s příznakem *read-only*.
Parametry: jméno makra

pmp.macro.Defn (defn)
Výpis definice makra. Vrací obsah makra. Pro vestavěná makra vrací prázdný řetězec.
Parametry: název makra

pmp.macro.Copydef (copydef)

Zkopíruje existující makro. Umožňuje definovat pro jedno makro více jmen. Tato dvě přiřazení jsou ale na sobě nezávislá; pozdější změna přiřazení makra k jednomu jménu nemá vliv na druhé jméno. Makro *copydef* umožňuje přejmenovat všechna vestavěná makra.

Parametry: nové jméno
původní jméno

pmp.macroCall (call)

Nepřímé volání makra. Umožňuje volat makra jejichž název neodpovídá regulárnímu výrazu pro identifikaci jmen maker a nelze je tedy volat přímo.

Parametry: jméno volaného makra
parametry předané volanému makru

pmp.macro.Pushdef (pushdef)

Provede předefinování makra. Obnovit původní definici je možné pomocí *popdef*.

Parametry: jméno makra
obsah makra

pmp.macro.Popdef (popdef)

Obnovení předchozí definice makra. Pokud makro již nemá žádnou další definici, pak je oddefinováno.

Parametr: jméno makra

pmp.macro.MacroTable (macros)

Makro pro manipulaci s tabulkou maker. Umožňuje provést import konfiguračního souboru a export současného stavu. Nemění makra s příznakem *read-only*.

Možné akce jsou: `export`, `import` a `replace` (nahrazení současného seznamu obsahem konfiguračního souboru).

Parametry: požadovaná akce
jméno souboru

pmp.macro.Halt (halt)

Ukončí práci makroprocesoru.

Parametry: typ zprávy zprávy (chyba, varování,…)
text vygenerované zprávy

pmp.Version

Vypíše informace o verzi makroprocesoru.

Nástroje pro lexikální analýzu

pmp.macro.lex.CEncoder

Zakóduje daný vstup do řetězce dle syntaxe jazyka C.

Parametry: vstupní text

pmp.macro.lex.CEscDecoder

Analyzátor řetězcových symbolů podle syntaxe jazyka C. Ve výchozím nastavení se používá pro zpracování speciálních symbolů v řetězcích.

Parametry: speciální symbol

pmp.macro.lex.JavaEscDecoder

Analyzátor řetězcových symbolů podle jazyka Java.

pmp.macro.lex.HTMLEntityDecoder

Analyzátor znakových entit z HTML/XML.

Parametry: znaková entita

Makra pro práci s řetězci

pmp.macro.str.SubStr

Vrací podřetězec daného řetězce

Parametry: řetězec
počáteční index vráceného podřetězce (počítáno od 0)
[maximální délka podřetězce]

pmp.macro.str.Length

Vrací délku zadaného řetězce.
Parametry: řetězec znaků

pmp.macro.str.Locate

Vrací pozici podřetězce v daném řetězci. Vrací prázdný řetězec, pokud neuspěje.
Parametry: hledaný podřetězec
prohledávaný text

pmp.macro.str.Compare

Porovnává dva řetězce. Pokud jsou stejné vrací `"1"`.
Parametry: první řetězec
druhý řetězec
[příznak `"ignore-case"`, jinak se porovnává s rozlišením velikosti písmen]

pmp.macro.str.LowerCase

Převede velká písmena v textu na malá.
Parametry: vstupní text

pmp.macro.str.UpperCase

Převede malá písmena v textu na velká.
Parametry: vstupní text

pmp.macro.str.Trim

Odstraní mezery z konce a/nebo začátku řetězce.
Parametry: text
režim (*LEADING*, *TRAILING*, *BOTH*)

pmp.macro.str.Match

Porovnává daný řetězec s regulárním výrazem. Vrací `"1"`, pokud řetězec odpovídá výrazu.
Parametry: řetězec
regulární výraz

pmp.macro.str.Replace

Nahradí všechny výskyty daného vzoru v textu zadanou náhradou.
Parametry: text
hledaný vzor (regulární výraz)
náhrada

pmp.macro.str.Rot13

Provede „šifru“ Rot13 na zadaný text.
Parametry: vstupní text

pmp.macro.str.MD5

Spočítá MD5 kontrolní součet ze zadaného řetězce.
Parametry: vstupní text

pmp.macro.str.Adler32

Spočítá kontrolní součet Adler32 pro zadaný text.
Parametry: vstupní text

pmp.macro.str.CRC32

Spočítá kontrolní součet CRC32 pro zadaný text.
Parametry: vstupní text

pmp.macro.str.Base64

Kóduje a dekóduje Base64.
Parametry: směr (*ENCODE* / *DECODE*)
text

pmp.macro.str.UUID

Vygeneruje náhodný UUID/GUID identifikátor.

pmp.macro.str.IsNumber

Vrací "1", pokud je zadaný text platným celým číslem podle specifikace Javy.
Parametr: text

pmp.macro.str.Chr

Vrací znak s daným kódem.
Parametry: kód znaku

pmp.macro.str.Ord

Vrací kód prvního znaku v daném řetězci.
Parametr: znak

Další pomocná makra

pmp.macro.math.Eval

Provede vyhodnocení zadaného výrazu. Vrací výslednou hodnotu. Umí vyhodnocovat základní matematické operace, včetně modulo (`"%"`), a disponuje sadou vestavěných funkcí (abs, signum, sin, cos, log,…). Detailní popis syntaxe je uveden v *Javadoc* dokumentaci této třídy.
Parametry: aritmetický výraz

pmp.macro.math.IfNotZero

Vyhodnotí zadaný výraz a podle výsledku vrátí hodnotu příslušného parametru.
Parametry: aritmetický výraz
výsledek v případě, že hodnota výrazu není nula
výsledek v případě, že hodnota výrazu je nula

pmp.macro.math.Min

Vrátí nejmenší číselnou hodnotu nalezenou mezi parametry. Pokud žádnou nenalezne, vrátí 0.
Parametry: vstupní hodnoty

pmp.macro.math.Max

Vrátí největší číselnou hodnotu nalezenou mezi parametry. Pokud žádnou nenalezne, vrátí 0.
Parametry: vstupní hodnoty

pmp.macro.math.Dec

Sníží číselnou hodnotu v definici makra.
Parametry: jméno makra (pokud neobsahuje číslo, vezme se 1)
hodnota o kterou se má snižovat (výchozí hodnota 1)

pmp.macro.math.Inc

Zvýší číselnou hodnotu v definici makra.
Parametry: jméno makra (pokud neobsahuje číslo, vezme se 0)
hodnota o kterou se má zvyšovat (výchozí hodnota 1)

pmp.macro.math.Random

Vrací náhodné číslo z daného rozsahu. Pokud je makro zavoláno s jedním parametrem, považuje se ten za horní mez.
Parametry: dolní mez (včetně); výchozí hodnota 0
horní mez (vyjma); výchozí hodnota 2^{31}-1

pmp.macro.math.Const

Vrátí hodnotu zadané konstanty (PI, SQRT_2, SQRT_3, apod).
Parametry: název konstanty (kompletní přehled viz Javadoc dokumentace této třídy).

pmp.macro.tools.Exec

Spustí program se zadanými parametry. Volání tohoto makra se nahradí výstupem volaného programu.
Parametry: název programu
parametry pro spouštěný program

pmp.macro.tools.GetProperty

Vrací hodnotu systémové proměnné (*system property*, viz dokumentace *java.lang.System*).
Parametry: jméno proměnné

pmp.macro.tools.SetProperty

Smaže nebo zruší obsah systémové proměnné.
Parametry: jméno
nová hodnota; pokud chybí, je proměnná smazána

pmp.macro.tools.Pause

Pozastaví běh makroprocesoru po danou dobu. Podporuje několik režimů:

SLEEP – prosté čekání (výchozí možnost)
WAIT – čekání na uplynutí intervalu a pak na stisk klávesy
INTR – čekání na uplynutí intervalu nebo na stisk klávesy

Poznámka: interakce s klávesnicí nefunguje pod Antem nebo pokud se čte vstupní program ze vstupu *stdin*.

Parametry: čas v milisekundách
režim

Makra z balíku *pmp.macro.java*

Tato makra umožňují psát jednoduché skripty pro manipulaci s Java objekty. Manipulace s objekty probíhá pomocí identifikátorů ve tvaru *{jméno třídy}@{číslo objektu}*. Tyto identifikátory jsou vraceny makry pro vytváření objektů a makra pro vyvolávání metod je očekávají na místě parametrů.

pmp.macro.java.Construct

Vytvoří novou instanci dané třídy.

Parametry: jméno třídy
parametry konstruktoru

pmp.macro.java.New

Vytvoří instanci wrapper objektu pro zadaný primitivní typ, nebo pole tohoto typů.

Parametry: jméno primitivního typu (v hranatých závorkách může následovat délka pole)
hodnoty...

pmp.macro.java.NewString

Makro pro jednodušší vytváření objektů typu String.

Parametry: textová konstanta

pmp.macro.java.Invoke

Zavolá metodu daného objektu, nebo statickou metodu dané třídy.

Parametry: jméno třídy nebo identifikátor objektu
jméno metody
parametry (identifikátory objektů)...

pmp.macro.java.Field

Vrací hodnotu, nebo nastavuje novou, daného pole specifikovaného objektu

Parametry: jméno třídy nebo identifikátor objektu
jméno pole
[nová hodnota (identifikátor objektu)]

pmp.macro.java.ToString

Zavolá metodu *toString()* daného objektu a vrátí výsledek.

Parametry: identifikátor objektu

pmp.macro.java.ToClassName

Vrátí jméno třídy daného objektu.

Parametry: identifikátor objektu

pmp.macro.java.InstanceOf

Vrací `"1"` pokud je daný objekt instancí dané třídy nebo rozhraní.

Parametry: identifikátor objektu
jméno třídy nebo rozhraní

pmp.macro.java.Cast

Změní typ daného objektu. Cílovým typem musí být předek třídy tohoto objektu, nebo implementované rozhraní.

Parametry: identifikátor objektu
jméno třídy nebo rozhraní

Přehled speciálních symbolů pro definice uživatelských maker

$$

Nahradí se znakem `$`.

$#

Vypíše počet parametrů jako dekadické číslo. Do výsledku se započítává i nultý parametr (jméno makra).

$*n*

Pokud je *n* číslice k se vypíše příslušný parametr 0 do 9 vypíše příslušný parametr. Numerická hodnota znaku se zjišťuje z Unicode tabulky pomocí metody *Character.getNumericValue(char)*, takže je možné zadat i jiné číslice než 0-9.

${*nnn*}

Nahradí se parametrem s číslem *nnn.*

${*nnn*-*mmm*}

Nahradí se parametry v rozsahu *nnn* až *mmm.*

$*

${*}

Nahradí se parametry makra (bez parametru nula) oddělenými znakem čárka.

${*makro*}

Nahradí se definicí daného makra.

Složené závorky ve speciálních symbolech je možné nahradit kulatými (např: `$(4-6)` místo `${4-6}`). V tomto případě dojde ke zpracování obsahu makrem (enkodérem) definovaným v direktivě *defined.encoder* a jednotlivé parametry budou odděleny řetězcem získaným z direktivy *defined.separator*.

Gramatika vstupního souboru

Gramatika, kterou se řídí syntaktický analyzátor při zpracovávání programu:

<input> ::= (<code-block> | <plaintext>)*

- vstupní neterminál

<plaintext> ::= <character>*

- text mimo kód je chápán pouze jako posloupnost znaků

<code-block> ::= <code-start> <code>* <code-end>

- kód (část vstupního programu, kde jsou identifikována volání maker

<code> ::= <string>

- řetězec znaků

<code> ::= <macro>

- volání makra

<code> ::= <code-token>

- posloupnost znaků, které nemají pro makroprocesor žádný speciální význam

<string> ::= <string-start> <string-element>* <string-end>

- řetězec znaků zpracovávaný makroprocesorem

<string-element> ::= <parser-token>

<string-element> ::= <string-token>

<string-element> ::= <character>

- elementy řetězce
- *parser-token* je dále zpracováván přiřazeným makrem (na rozdíl od *string-token)*

<macro> ::= <macro-name> <parameters>

- volání makra s parametry

<macro> ::= <macro-name>

- volání makra bez parametrů
- toto pravidlo platí jen pokud není nastavena direktiva *macro.args.required*

<parameters> ::= <opening-bracket> <param-list> <closing-bracket>

<param-list> ::=<param> <param-separator> <param-list>

<param-list> ::= <param>

<param> ::= <code>*

- parametry volaného makra

Konfigurační soubory

Konfigurační soubory umožňují provést snadnou konfiguraci makroprocesoru. Mají formát XML a jejich struktura je definována pomocí pomocí DTD. Výchozí nastavení je umístěno v jar archivu makroprocesoru v souboru *pmp/config/config.dtd.*

Struktura konfiguračního souboru

Kořenový element má jméno *config*. Jeho podelementy jsou:

define
: Definice makra. Jméno je v atributu *name*. Atribut *readOnly* určuje zda bude možné definici makra změnit.

class
: Načtení makra z *.class* souboru. Jméno je v atributu `name`.

import
: Přidá konfigurační data ze souboru zadaného v atributu *url*.

input
: Určí vstupní program makroprocesoru.

library
: Umístění knihovny (jar archiv) s vlastními makry v class souborech.

Parametry příkazové řádky

Makroprocesor se dá pustit z příkazové řádky pomocí parametru *-jar* interpreteru Java.

syntaxe volání:

```
java –jar pmp.jar [přepínače] [--] [soubory]
```

-o {soubor}
Specifikuje výstupní soubor.

-d {úroveň}
Nastaví úroveň výpisu ladících informací (možno zadat číselně nebo předdefinovanou konstantu):
NODEBUG (0) – nic se nevypisuje (výchozí nastavení)
MACRO_CALLS (10) – výpis volaných maker
MACRO_RESULTS(15) – výpis výsledků volání maker
EVERY_TOKEN (20) – výpis jednotlivých lexikálních elementů

-D {makro=definice}
Definice makra. Do seznamu maker se vloží makro se zadaným jménem a obsahem.

-m {makro=třída}
Načtení třídy s makrem. Do seznamu maker se k danému jménu přiřadí instance zadané třídy.

-c {soubor}
Načte konfiguraci ze zadaného souboru.

-v
Vypíše verzi makroprocesoru a skončí.

-ie {kódování}
Definice kódování vstupních souborů.

-oe {kódování}
Definice výstupního kódování

-gui
Spustí grafické rozhraní makroprocesoru.

-version
Zobrazí číslo verze a skončí.

-help
Vypíše popis syntaxe volání a seznam přepínačů.

--
Značí konec přepínačů. Následující parametry jsou považovány za jména souborů (i pokud začínají pomlčkou).

Ant Task

Úloha *PMP* slouží ke spouštění makroprocesoru z Antu.

Definice úlohy

Import do Ant se provede pomocí příkazu *taskdef, např:*

```
<taskdef resource="pmp/ant/pmp.properties"
     classpath="pmp.jar" />
```

Parametry

Atribut	*Popis*
debug	určuje množství vypisovaných ladících informací
src	zdrojový adresář
dest	cílový adresář
srcEncoding	kódování zdrojových souborů
destEncoding	výstupní kódování
defaultConfig	určuje, zda má se použít výchozí konfigurace
failonerror	určuje, zda se má ukončit zpracovávání souborů, pokud se narazí na chybu
importPrefix	prefix pro import proměnných (properties) z Antu

Podelementy

Tato úloha je založená na třídě FileSet a je možné používat podelementy této úlohy (*include*, *exclude*, apod...).
Pro transformaci jmen souboru je možné vložit *Mapper*.

Element config

Tento element odpovídá kořeni konfiguračního souboru, tak jak je popsán výše.

Element export

Tento element určuje, která makra se mají exportovat jako Ant property po skončení práce makroprocesoru.

Podelement *macro* exportuje dané makro a podelement *classname* exportuje název třídy, která odpovídá danému makru.

Jméno makra určeno textovým obsahem elementu.

Atribut	*Popis*
name	název, pod jakým se má makro exportovat
mode	určuje, jak se má export provést: *first* – exportuje se první definovaná hodnota *last* – exportuje se poslední definovaná hodnota *concat* – exportují se všechny hodnoty spojené do jednoho řetězce *sum* – exportuje se součet číselných hodnot

Podelement *all* zajistí export všech maker.

Příklad použití:

```
<pmp src="src"
        dest="${build}"
        srcEncoding="utf-8" >
     <include name="*.pmp" />
     <mapper type="glob" from="*.pmp" to="*.txt" />
</pmp>
```

Zavolá makroprocesor na všechny soubory v adresáři *src* s příponou *pmp* a uloží výstup do adresáře definovaného proměnnou *build*. Zdrojové soubory by měli mít kódování *utf-8*. Zapisované soubory budou mít příponu *txt* a budou ve výchozím kódování dané platformy. Struktura podadresářů zůstane zachována, ale ve výstupu se vytvoří pouze adresáře, které obsahovali nějaký soubor **.pmp*.

Obsah CD

název	***popis***
/AdobeReader7	prohlížeč dokumentů PDF
/Ant-1.6.5	Ant 1.6.5
/dokumenty/Bc_prace	tato práce
/dokumenty/Dokumentace	dokumentace k ročníkovému projektu
/dokumenty/PMPTask	manuál k úloze PMPTask pro Ant
/dokumenty/Specifikace	specifikace ročníkového projektu
/jdk-1.5	Java Development Kit 1.5
/pmp	vlastní makroprocesor

Poznámka: Dokumenty na přiloženém CD jsou obvykle ve více formátech. Původní je vždy *OpenDocument Text* (**.odt*), ostatní vznikly exportem.