Fakulta informačních technologií

Vysoké učení technické v Brně

Příručka pro studenty předmětu Formální jazyky a překladače

Jak na projekt

(IFJ07)

Zbyněk Křivka

Roman Lukáš

Lukáš Rychnovský

Projekt FRVŠ 673/2007/G1

Úvod

Následující příručka pro studenty předmětu *Formální jazyky a překladače* (IFJ) a *Algoritmy* (IAL) slouží pro stručný úvod, jak je možné implementovat překladač a interpret, což je hlavní náplní společného týmového projektu do obou těchto předmětů bakalářského studijního programu. K tomuto dokumentu je také nachystána prezentace, která s následujícím obsahem seznámí studenty na demonstračních cvičeních z IFJ.

Jelikož se jedná pravděpodobně o první překladač, který studenti programují, věříme, že jim přijde vhod tato krátká příručka a souhrn doporučení, jak při implementaci projektu postupovat a na co si dát během řešení pozor.

Všechny zde uvedené rady a návody jsou samozřejmě nezávazné a netvrdíme, že jsou nejlepší možné, ale snažili jsme se vytvořit rovnováhu mezi kvalitou a jednoduchostí implementace doporučovaných rad.

Obecnými otázkami projektového řízení a týmové spolupráce se přinejmenším teoreticky zabývají i některé ostatní předměty, proto pouze pro zopakování a souhrn základních informací se odkažme na příručku „Práce v týmu“. V této příručce se potom zaměříme na konkrétní tipy a rady s implementací se zaměřením na projekt se zadáním **jazyka IFJ07**.

Základní informace

Cíle a hlavní účel projektu

Hlavním cílem je, aby studenti pomocí vypracování projektu porozuměli a pochopili základům tvorby překladačů a interpretů. Při té příležitosti by se měli naučit také týmové spolupráci, která je v dnešní době nepostradatelnou výbavou moderního programátora.

V rámci dodatečných požadavků na absolvování projektu si vyzkouší ústní obhajobu své práce a naučí se tak své nápady prezentovat. Kromě implementace abstraktních datových struktur a operací nad nimi si studenti také ověří znalosti z programovacích jazyků jako je rekurzivní volání funkcí a podobně.

Struktura projektu

Hlavní skeleton struktury projektu obsahuje tři nejdůležitější části:

1. lexikální analyzátor (scanner)
2. syntaktický a sémantický analyzátor (parser)
3. interpret

Samostatné úkoly na projektu

Dříve než se budeme podrobněji věnovat jednotlivým částem, vyjmenujeme úkoly na projektu, které tyto hlavní části doplňují a nebo ještě dále dekomponují a lze je rozdělovat mezi členy týmu, protože jejich implementace je až na domluvu společného rozhraní nezávislá.

Detailnější rozvahu o způsobu dělení práce mezi řešiteli projektu najdete v příručce o týmové práci.

- vedení projektu (kontrola plnění plánu, kontrola kvality, oficiální komunikace při zastupování celého týmu, spravedlivé rozdělování úkolů, rozdělování bodů, atd.)

- generování programátorské dokumentace, její manuální doplnění, tvorba popisů rozhraní, na kterých se tým během konzultací dohodl, vytvoření případné uživatelské dokumentace
- návrh rozhraní mezi jednotlivými částmi překladače a interpretu
- lexikální analyzátor (neboli scanner) založený na konečném automatu nebo regulárních výrazech
- návrh instrukční sady vhodné pro reprezentaci přeloženého programu a následnou interpretaci
- implementace potřebných abstraktních datových typů, náročnějších datových struktur a operací s nimi (např. lokální/globální tabulky symbolů), programování vestavěných funkcí (např. řazení)
- návrh gramatiky a implementace syntaktického analyzátoru (bez zpracování výrazů; budování abstraktního syntaktického stromu včetně případných sémantických akcí; většinou doporučujeme nebo přímo diktujeme využít analýzu shora-dolů)
- syntaktický analyzátor pro výrazy (většinou doporučujeme precedenční syntaktickou analýzu zdola-nahoru)
- interpret
- vytvoření testovacích příkladů programů včetně vstupů a očekávaných výstupů pro potřeby ladění a testování jednotlivých částí aplikace i aplikace jako celku
- sestavování výsledné aplikace a její testování na vlastních i zveřejněných příkladech (po dodání všech částí s definovaným rozhraním)

Projekt lze v případě excelentního týmu obohatit i o bonusové vlastnosti, jež jsou nad rámec požadavků na projekt a většinou jsou hodnoceny bonusovými body (až 5 bodů) nad maximální počet bodů (20 bodů implementace, 5 bodů dokumentace, 5 bodů prezentace) za projekt.

- zotavování z chyb
- detailnější a intuitivnější chybové hlášky (rozlišování lexikální, syntaktické, sémantické chyby nebo chyby během interpretace)
- zkrácené vyhodnocování logických výrazů
- optimalizace instrukční sady, počtu instrukcí a případně interpretu
- kontrola existence návratové instrukce ve všech možných větvích definované funkce
- kontrola, že proměnná byla definována (byla jí přiřazena nějaká hodnota) před použitím proměnné například na pravé straně přiřazovacího příkazu nebo obecně v aritmetickém či logickém výraze
- umožňování smysluplných implicitních a případně i explicitních konverzí datových typů ve výrazech

Rozdělení úlohy mezi členy týmu

Tvorbu překladače s interpretem lze rozdělit na několik relativně nezávislých částí, na kterých lze pracovat samostatně a pouze s občasnými týmovými schůzkami, kdy se dořeší vzájemné rozhraní modulů a další problémy, na které řešitelé v průběhu návrhu a

implementace narazí. Vycházet můžete například z výčtu úloh v předchozí podkapitole a z příručky pro studenty o práci v týmu.

Lexikální analyzátor

Lexikální analyzátor má za úkol rozpoznávat jednotlivé lexémy (lexikální jednotky) zadaného programu, který je napsán v zadaném programovacím jazyce (požadavky například na tvary čísel, řetězců apod. jsou uvedeny v zadání projektu). Z lexikálních jednotek potom vytvoříme korespondující tokeny, které kromě typu lexému obsahují důležité doplňující informace pro sémantickou analýzu a interpretaci (např. jméno identifikátoru, konkrétní hodnotu konstanty apod.).

Nejpřímočařejší implementace lexikálního analyzátoru je pomocí konečného automatu (v jazyce C se nejčastěji použije řídící konstrukce `switch` spolu s řídící proměnnou, která uchovává aktuální stav automatu). Analyzátor musí být tedy schopen rozpoznat a vrátit na požádání jeden token, který následuje za předešlým již načteným tokenem. Implementace je prováděna většinou funkcí, která vrací strukturu reprezentující načtený token (např. `get_token()`). Některé metody syntaktické analýzy mohou požadovat navrácení tokenu zpět do proudu tokenu a jeho znovunačtení při dalším požadavku na nový token. Je-li toto chování nutné, lze k tomu využít vyrovnávací paměť pro jeden token a informativní stavová proměnná říkající, zda vyrovnávající paměť (implementovaná obyčejnou proměnnou) obsahuje platná data nebo nikoli a pak se provede načtení nové lexikální jednotky a vytvoření nového tokenu.

V případě obecnější implementace lze lexikální analýzu založit na regulárních výrazech, jako to dělá například generátor lexikálních analyzátorů, `lex` (nebo jeho modernější varianta `flex`).

Syntaktická analýza

Syntaktická analýza, která je většinou úzce provázána i se sémantickou analýzou a vytvářením abstraktního syntaktického stromu či přímo posloupnosti tříadresných instrukcí, je doslova srdcem překladače a zároveň jeho nejsložitější částí.

Pro mírně zjednodušení jsou v posledních letech zadání koncipována tak, aby bylo možno syntaktickou analýzu rozdělit na dvě úrovně:

A) syntaktická analýza jazykových konstrukcí (deklarace/definice proměnných a funkcí, řídící/přiřazovací příkazy apod.), která mívá jednoduchou strukturu popsatelnou konečným automatem. V praxi se pak většinou setkáte se složitějšími jazyky, které vyžadují některou z pokročilejších typů syntaktické analýzy (LL(1) syntaktická analýza, LR syntaktická analýza atd.).

B) syntaktická analýza výrazů (aritmetických i logických), která již konečným automatem popsat nejde. Pro tuto část analýzy je třeba precedenční syntaktický analýza (typ analýzy zdola-nahoru, která potřebuje vytvořit řídící tabulku, jež má pro konkrétní jazyk v jednotlivých buňkách statické hodnoty)

Úkolem syntaktické analýzy tedy je zkontrolovat syntaxi vstupního programu, provést korespondující sémantické akce a vygenerovat tříadresný kód (tj. posloupnost tříadresných instrukcí), jež bude následně interpretován interpretem.

Interpret

Interpret je poslední část projektu, která na vstupu dostává program reprezentovaný posloupností instrukcí, které jste si sami navrhli (doporučujeme instrukční sadu založenou na tříadresných instrukcích). Dalším vstupem je samotný vstup programu (například zadané číslo pro výpočet faktoriálu apod.). Interpretace pak provede podle instrukcí zpracování vstupu programu a vygeneruje výstup (jinak by program neměl valný smysl, ale pro testovací účely mohou být použity i programy, které neberou ani vstup, ani negenerují/nevypisují žádný výstup).

Implementace interpretu většinou ve smyčce prochází seznamem tříadresného kódu, který se vykonává lineárně, dokud nenarazí na skokovou instrukci, která může ukazatel na aktuální instrukci k provedení přesunout kamkoli v celé posloupnosti instrukcí, které program reprezentují. Vykonávání těchto instrukcí vyžaduje různé pomocné struktury jako zásobníky a různé tabulky symbolů (viz dále). Součást návrhu interpretu je i návrh optimální instrukční sady, který závisí na povolených jazykových konstrukcích, které má váš překladač zpracovávat a generovat pro ně instrukce. Dobře zvolená a vymyšlená instrukční sada může ušetřit hodně programování.

Tabulka symbolů

Jelikož je většina programovacích jazyků podle Chomského hierarchie jazyků kontextových, nedostačuje na jejich zpracování pouze čistý syntaktický analýza, která popisuje většinou jen podmnožinu bezkontextových jazyků. Na pomoc si je třeba vzít dodatečné mechanismy jako sémantickou analýzu a úložiště dat potřebných pro správnou sémantickou analýzu. Tímto úložištěm dat bývají nejčastěji různé tabulky symbolů.

Tabulka symbolů je struktura obsahující množinu nebo i multimnožinu dvojic (klíč, hodnota). Klíčem bývá zpravidla identifikátor (ať už přímo uvedený v programu nebo pomocný například pro dočasné pomocné proměnné). Hodnotou jsou pak nejrůznější potřebné informace i data vztahující se k danému identifikátoru (viz níže). V rámci předmětu IAL byl požadavek na implementaci efektivní netriviální tabulky symbolů. Protože je to struktura, ve které se zpravidla častěji vyhledávají záznamy než že by se modifikovaly (přidávaly, odebíraly, měnily), tak jsou různé varianty zadání, kde studenti implementují tabulky symbolů pomocí:

A) hashovací tabulky
B) binárních vyhledávacího stromu
C) AVL stromu

Detaily ohledně těchto datových vyhledávacích struktur naleznete v oporách k předmětu IAL.

Hodnoty pro daný klíč obsahují různá data v závislosti na typu tokenu, kterému identifikátor (klíč) odpovídá:

- **Proměnná**: Kromě typu proměnné potřebujeme často uchovávat také index proměnné ve struktuře, která reprezentuje paměťové úložiště pro proměnné (v reálné implementaci bude pravděpodobně index nahrazen ukazatelem do paměti).
- **Identifikátor funkce** (jméno funkce): Zde je třeba si zapamatovat informaci o typech všech vstupních i výstupních parametrů a typ samotné funkce (tedy jaký typ dat funkce vrací při dokončení svého běhu), dále je také potřeba vědět, zda již

byla funkce definována či nikoli kvůli možnosti kontroly kolizí dvou definic stejných funkcí nebo naopak chybějící definice nějaké používané funkce.
- **Návěští**: Návěští označuje identifikátorem cíl skoku (ať již podmíněného či nikoliv). Z toho důvodu je třeba si pamatovat, zda bylo nalezeno návěští jako cíl skoku a mezi kterými instrukcemi, nebo zda byl pouze nalezen skok na nějaké návěští, jehož existenci jsme zatím nezaregistrovali a bude potřeba ji zkontrolovat a zajistit.

Hlavní implementačním doporučením je vytvořit dvě instance tabulky symbolů: lokální a globální. Lokální bude obsahovat informace nutné při analýze definice funkce a globální všechny ostatní deklarace s globální viditelností (v celém programu). Zda budeme s hlavní funkcí pracovat jako s význačnou funkcí s vlastní lokální tabulkou symbolů nebo pracující s globální tabulkou je již závislé na konkrétním implementačním rozhodnutí.

Deklarace globálních proměnných

Při nalezení globální deklarace proměnné provedeme kontrolu, zda již nebyla deklarována dříve a pokud ne, tak provedeme vložení nového záznamu do globální tabulky symbolů. Kromě tohoto bude potřeba také vygenerovat pro interpret instrukci, která provede alokaci globálního datového bloku pro tuto proměnnou. Velikost a struktura bloku samozřejmě závisí na typu této proměnné. Hodnota Indexu nebo chcete-li pozice ukládaná do tabulky symbolů k této proměnné potom bude odkazovat na patřičný globální blok, který bude během interpretace obsahovat data, jež proměnná obsahuje.

Implementace souboru globálních datových bloků musí zohledňovat různé podporované datové typy a předem neznámý počet globálních proměnných, protože jejich počet může být v každém programu jiný.

Deklarace funkcí

Podobně jako v případě proměnných i u deklarace funkcí musíme provést kontrolu, zda již funkce nebyla deklarována či dokonce definována. Pokud nebyla, tak vytvoříme nový záznam do globální tabulky symbolů, kde si poznačíme, že jsme nalezli deklaraci (někde dále předpokládáme existenci definice, pokud již nebyla načtena dřív, ale to by bylo možné pouze v jazyce, kde nevyžadujeme, aby deklarace funkce/proměnné předcházela jejímu definici a použití). Uložíme si také seznam typů parametrů (identifikátory formálních parametrů pro nás v tomto jazyce nejsou klíčové, protože je pevně dáno jejich pořadí) a typ návratové hodnoty funkce.

Jako tip vám navrhujeme pro uložení typů parametrů a návratové hodnoty využít řetězec, kde první znak bude popisovat typ návratové hodnoty funkce a ostatní znaky budou popisovat popořadě jednotlivé parametry. Například `void f(int, string, double)` bude mít u typů parametrů řetězec „`visd`“, protože `v` = `void`, `i` = `int`, `s` = `string` a `d` = `double`.

Definice funkcí

Definice funkce je část programu, kde kromě hlavičky funkce uvedeme také její tělo, tedy popíšeme algoritmus, který funkce vykonává nad parametry a jehož výsledek pak vrací. Definice funkce se většinou skládá z hlavičky, deklarace lokálních proměnných,

těla funkce a konce funkce. Někdy jsou poslední tři jmenované části obsaženy všechny v těle funkce.

Definice funkce a deklarace funkce jsou provázány přes identifikátor funkce a v některých jazycích i přes výčet parametrů (buď záleží pouze na počtu, nebo i na typech parametrů).

Hlavička

V hlavičce se často opakují informace z deklarace funkce a někdy jsou doplněny například o jména formálních parametrů, která nejsou v tom případě v deklaraci funkce vyžadována. Pokud již máme definici předcházející deklaraci funkce, tak definice musí odpovídat počty i typy svých parametrů i návratovým typem. Po zkontrolování můžeme provést záznam o definici funkce do globální tabulky symbolů, kde již by měl být záznam o deklaraci této funkce.

Před všechny úkony, které funkce provádí, vložíme návěští pro skok, abychom mohli funkci volat (resp. v pojmech interpretu „skočit na začátek této funkce“).

Dále se musíme vypořádat s formálními parametry uvedenými v hlavičce, které se v těle funkce musí chovat v podstatě jako běžné proměnné. Provedeme tedy vložení jednotlivých parametrů jako proměnných do lokální tabulky symbolů.

Lokální proměnné

Při deklaraci nových lokálních proměnných uvnitř funkcí je potřeba provést sémantickou kontrolu, zda je název unikátní a tedy se neshoduje s nějakou již deklarovanou lokální proměnnou nebo formálním parametrem funkce, které se jako lokální proměnné tváří.

Jednotlivé proměnné vložíme podobně jako parametry funkce do lokální tabulky symbolů spolu s dalšími potřebnými informacemi (typ, pozice a případně další). Pro interpret je třeba také vygenerovat instrukce, které alokují pro proměnnou resp. její typ potřebný lokální datový blok.

Konec bloku (funkce nebo programu)

Bloky v jazyce IFJ07 jsou pouze funkce (tvoří lokální bloky) a celý program, který tvoří blok globální. Na konci bloku je potřeba provést úklid alokované paměti pro lokální proměnné a další pomocné datové struktury následovně:

A) **Konec funkce**: Nejprve provedeme sémantickou kontrolu, zda všechny odkazované návěští byly ve funkci nalezeny (definovány), aby nedošlo k situaci, že funkce se v některé její části snaží skočit na neexistující návěští uvnitř této funkce (přes návěští nepovolujeme odskoky mimo funkci, abychom nebořili strukturovanost programovacího jazyka; tedy každé návěští je pouze lokálního charakteru; návěští globálního charakteru jsou začátky funkcí programu). Dále je potřeba vyprázdnit lokální tabulku symbolů a vygenerovat instrukce, které dealokují datové bloky, jež byly vyhrazeny pro lokální proměnné a parametry. Je potřeba zajistit také práci s návratovou hodnotou, což bude diskutováno u volání funkce (či volání podprogramu interpretem).

B) **Konec programu**: Je potřeba opět zkontrolovat, zda byly všechny deklarované funkce i definované a zda byla nalezena funkce `main` se správnými parametry. Analogicky ke konci funkce je u konce programu nutné vyprázdnit globální tabulku symbolů a vygenerovat pro interpret instrukce dealokující datový blok

globálních proměnných. Poznamenejme, že zdrojový text programu nemusí vždy nutně končit koncem funkce `main`, ale i libovolné jiné či jiné globální jazykové konstrukce.

Generování kódu

Nyní si popišme sémantické kontroly a generování instrukcí pro základní programové příkazy a konstrukce (přiřazení, nepodmíněné a podmíněné skoky, návrat z funkce, zpracování výrazů a volání funkce).

Poznamenejme, že při kontrolách kompatibility typů výrazů je třeba brát v úvahu i potencionální implicitní či explicitní typové konverze, které se v praxi často používají. Jak je to v projektu, záleží na konkrétním zadání.

Přiřazení:

- `id := <výraz>;`

Po provedení sémantické kontroly, zda typ výrazu a typ proměnné *id* odpovídají, je nutno zkontrolovat také to, že proměnná *id* již byla dříve deklarována (nemusela však ještě být definována). Následně je nagenerována instrukce, která kopíruje hodnotu výrazu do paměti alokované pro proměnnou *id* (pro ukládání hodnot do proměnné *id*). Zdůrazněme, že do paměti je nanejvýš vhodné přistupovat pouze pomocí indexů (či pozic), které známe z tabulky symbolů pro daný identifikátor proměnné. Dále nesmíme zapomenout rozlišovat lokální a globální proměnné, jejichž metainformace se nachází v různých tabulkách symbolů.

Návěští, podmíněný a nepodmíněný skok:

- `id:`

V případě deklarace nového návěští je třeba provést sémantickou kontrolu, že návěští stejného jména ještě nebylo použito v dané funkci a je třeba zajistit jednoznačné označení návěští na instrukční pásce celého programu, protože stejné návěští se může vyskytnout ve více různých funkcích, ale nesmí dojít k záměně! Pak se provede případné zaregistrování identifikátoru jako návěští do odpovídající tabulky symbolů (uvnitř funkce většinou lokální).

- `goto id;`

Zkontrolujeme, zda již známé místo deklarace návěští id, na které chceme tímto nepodmíněným skokovým příkazem skočit. Pokud ještě návěští nebylo deklarováno, je třeba si poznačit, že její deklaraci očekáváme nejpozději na konci funkce, protože jazyk IFJ07 nepodporuje skoky mimo funkci (funkci lze opustit pouze příkazem return jako ve většině strukturovaných programovacích jazyků). Nageneurujeme odpovídající instrukci pro nepodmíněný skok na danou lokaci na instrukční pásce.

- `if <výraz> goto id;`

Analogicky k nepodmíněnému skoku, ale nejprve je třeba zajistit vyhodnocení podmínky a pak teprve zajištění skoku či vynechání skoku.

Proč je nutné vyžadovat jednoznačnost názvu návěští z hlediska globální úrovně? Jelikož je celý program pro interpret již reprezentován pouze jedinou dále nestrukturovanou sekvencí instrukcí, tak je třeba mít identifikátory cílů skoků (tj. návěští) unikátní v rámci

celé sekvence (tedy celé instrukční pásky). Jedno z možností, jak toho dosáhnout, je označovat všechny návěští předponou odpovídající identifikátoru funkce, ve které se návěští vyskytlo (pokud samozřejmě jazyk nepovoluje, aby návěští bylo i mimo funkce, protože pak bychom se museli uchýlit k dalším trikům jako využívání nepovolených znaků v interních pojmenováních návěští a podobně).

Návrat z funkce:

- `return <výraz>;`

  Opět provedeme kontrolu shody typu výrazu, jehož výsledek vracíme, a typu návratové hodnoty funkce (viz deklarace dané funkce). Nyní musíme uložit hodnotu výrazu, jehož výsledek má funkce vracet, na dané místo v lokálních datech (viz volání funkce, kde se zavolané funkci tato pozice vyhrazuje a sděluje její umístění).
- `return;`

  Typ návratové hodnoty funkce musí být void, jinak hlásíme sémantickou chybu.

Zbytek postupu je již shodný pro obě varianty návratu z funkce.

Z lokálního datového bloku typu „zásobník“ zjistí funkce, na jaké místo na instrukční pásce se má po ukončení této funkce skočit (většinou se jedná o následníka *N* instrukce, která tuto funkci volala). Z lokálního datového bloku se pak odstraní všechny proměnné, které byly v této funkci použity (deklarovány) a skočí se na instrukci *N*.

Po tomto příkazu vždy následuje konec funkce, takže se uplatní i pravidla platná pro konec bloku, případ A.

Zpracování výrazů

Zpracování výrazu zaštiťuje v našem návrhu i volání funkcí, a jelikož se jedná o rozsáhlejší pasáž, tak jí věnujme následující podkapitolu.

Pro aritmetické i logické výrazy je nejprve nutno vytvořit reprezentující gramatiku, kterou využijeme při konstrukci syntaktického analyzátoru. V projektu IFJ07 jsou vyžadovány především infixové binární operátory (aritmetické i relační) a je doporučeno výrazy zpracovávat precedenční syntaktickou analýzou. Navíc bylo také doporučeno se i na zavolání funkce dívat jako na výraz, který odpovídá navrácené hodnotě této funkce.

Po vytvoření kostry gramatiky podle předchozích požadavků získáváme tvar pravidel bezkontextové gramatiky (z důvodů zkrácení pracuje pouze se dvěma operátory op_1 a op_2):

1. $E \rightarrow id()$
2. $E \rightarrow id(E)$
3. $E \rightarrow id(L)$
4. $L \rightarrow L, E$
5. $L \rightarrow E, E$
6. $E \rightarrow id$
7. $E \rightarrow const$
8. $E \rightarrow (E)$

9. $E \rightarrow E\ \mathrm{op}_1\ E$
10. $E \rightarrow E\ \mathrm{op}_2\ E$

Konstrukce tabulky pro precedenční syntaktickou analýzu je důkladně popsána na přednáškách, v oporách a dokonce i na demonstračních cvičeních, takže ji budeme považovat za zvládnutou.

Jediný rozdíl oproti přednášce je v zahrnutí volání funkce do gramatiky zpracovávající výrazy a tudíž je potřeba v precedenční tabulce řešit i terminální symboly jako čárka (,) a specifický vztah *id* a levé závorky ()) zaviněné pravidlem 2 a 3.

Operátor čárka má význam oddělovače parametrů volané funkce. Protože parametry zpracováváme zleva doprava, tak má operátor čárka (viz pravidla 4 a 5) levou asociativitu. Jelikož navíc chceme mít možnost jako parametry uvádět celé výrazy a nikoli pouze identifikátory/konstanty, tak je třeba dát čárce nejmenší prioritu ze všech binárních operátorů (zde značeny op_1, op_2, atd.).

Tímto jsme dořešili téměř všechny nové buňky oproti tabulce na přednášce. Jednou z posledních problematických buněk je na řádku „*id*“ a sloupci „(“, což odpovídá blížící se aplikaci pravidla 2 nebo 3. Ti bystřejší pravděpodobně již tuší, že tuto buňku vyplním rovnítkem (=), které symbolizuje případ, kdy pravidlo na pravé straně obsahuje více terminálů a tudíž ještě nevyžadujeme provádění redukce a ani nechceme pravidlo roztříštit na více redukcí (nepoužijeme tedy < ani >). Analogický případ lze vidět i u pravidla 8 a buňky v tabulce na řádku „(“ a sloupci „)“.

Nyní projděme jednotlivá pravidla či podskupiny pravidel pracujících s výrazy a popišme si korespondující sémantické kontroly a akce:

Zpracování identifikátoru: $E \rightarrow id$
Po kontrole deklarace proměnné *id* je nutno přiřadit atributu nonterminálu *E* index proměnné *id* a její typ, což asi nejpohodlněji provedeme navázáním nonterminálu *E* na odpovídající záznam v odpovídající tabulce symbolů (je třeba si uvědomit, že *id* může být lokální i globální proměnná).

Zpracování konstanty: $E \rightarrow const$
Toto pravidlo je analogické k předchozímu s tím rozdílem, že je potřeba navíc na začátku vytvořit novou lokální dočasnou/pomocnou proměnnou typu, který specifikuje konstanta *const*. Tuto dočasnou proměnnou potom definujeme/inicializujeme hodnotou *const*. Nagenerujeme tedy všechny potřebné odpovídající instrukce a dále řešíme jako předchozí případ zpracování identifikátoru.

Zpracování výrazu v závorkách: $E_1 \rightarrow (E_2)$
Zde není provádět žádné speciální akce, ale pouze zkopírovat atributy instance pravého nonterminálu *E* (značen E_2) do atributů levé instance nonterminálu *E* (značen E_1).

Zpracování aplikace binárního operátoru (zde infixového): $E_1 \rightarrow E_2\ \mathrm{op}_n\ E_3$

Nejprve je třeba zkontrolovat, zda je operace op_n povolena nad danými datovými typy, které jsou reprezentovány symboly E_2 a E_3 (popřípadě zda lze provést implicitní konverze tak, aby operace op_n povolena byla). Nechť symbol t označuje typ výsledku operace.

Nyní nagenerujeme novou lokální pomocnou/dočasnou proměnnou typu t a atributu nonterminálu E_1 přiřadíme její index a typ. Dále vygenerujeme instrukci, která provede danou operaci op_n nad operandy E_2 a E_3 (indexy odpovídajících proměnných jsou uloženy právě v atributech operandů E_2 a E_3).

Zpracování volání funkce ve výrazu:
$E \rightarrow id(), E \rightarrow id(E), E \rightarrow id(L), L \rightarrow L, E, L \rightarrow E, E$
Volání funkce je asi nejsložitější konstrukcí aritmeticko-logických výrazů, které se v zadáních projektů vyskytují.

Pravidla jsou koncipována tak, aby bylo možno volat funkce s libovolným počtem parametrů. Jako domácí cvičení si dejme za úkol prozkoumat, proč je potřeba zvlášť jednat v pravidlech s jednoparametrickou funkcí a víceparametrickou.

Při postupném zpracovávání parametrů resp. výrazů, které odpovídají jednotlivým parametrům a jsou odděleny čárkami, si pamatujeme typy jednotlivých parametrů, abychom je mohli na závěr zkontrolovat se seznamem očekávaných typů parametrů podle deklarace případně definice funkce.

Následně provedeme vygenerování instrukcí, které postupně vloží na zásobník, jež reprezentuje lokální data, následující informace:

- aktuální pozici instrukce (resp. pozici následníka instrukce) na instrukční pásce, která bude potřeba při návratu z funkce;
- index/pozici pro proměnnou, do které bude uložen výsledek funkce (návratová hodnota); Nezapomeňme si uvědomit, že tato hodnota musí být přístupná i po opuštění funkce (a tedy po dealokaci jejího lokálního zásobníku), aby k ní mohla přistupovat funkce, která dokončenou funkci volala;
- hodnoty jednotlivých výrazů odpovídajících parametrům volané funkce (většinou se bude jednat o proměnné, jejichž identifikátor bude odpovídat patřičnému formálnímu parametru funkce a jejichž hodnota byla dosazena právě z výrazu předaného funkci na místě daného parametru) do lokální datové části.

Nakonec nagenerujeme instrukce pro nepodmíněný skok na začátek volané funkce, která má k tomu vygenerované odpovídající návěští (Opět pozor na nutnost globální unikátnosti identifikátorů návěští).

Interpret – příklad volání podprogramu

Interpret je poslední část aplikace, která dostává od překladače instrukční pásku, kterou podle programu na této instrukční pásce a podle vstupů algoritmicky provádí.

Nyní si uveďme pouze krátký příklad provádění výrazu volání funkce (resp. podprogramu). Hlavní body obsahují interpretované příklady tříadresných instrukcí vygenerovaných pro volání funkce *f*, její provedení a dokončení. Tyto body jsou pak dále rozepsány do několika kroků, které interpret provádí:

1) `[PUSH param],[CALL f,,ret_addr]`:

   a. Označit současný vrchol zásobníku jako vrchol zásobníku funkce a uložit předchozí vrchol zásobníku jako nadřazený datový zásobník.
   b. Vložit parametry volání funkce na zásobník.
   c. Uložit návratovou adresu instrukční pásky (datová struktura obsahující sekvenci instrukcí, které reprezentují celý program a kterou postupně interpret zpracovává).
   d. Uložit adresu pro uložení návratové hodnoty do nadřazeného datového zásobníku (pokud má tedy funkce návratovou hodnotu; tedy není typu `void`).
   e. Naalokovat místo pro lokální proměnné
2) … `instrukce těla funkce` …:
   a. Vykonat instrukce v těle funkce (nejčastěji algoritmus zpracovávající vstup a generující/vracející výstup).
3) `[RETURN value,,ret]`:
   a. Uložit návratovou hodnotu na adresu v nadřazeném datovém zásobníku (pozici jsme si programově v interpretu poznačili v bodě 4).
   b. Obnovit nadřazený datový zásobník do současného vrcholu zásobníku, čímž se uvolní lokální proměnné (resp. již na ně nebudu schopen se odkazovat a paměť budu moci alokovat k jiným účelům).
   c. Vrátit se na uloženou pozici v instrukční pásce, čemuž odpovídá tomu, že se bude pokračovat od místa, kde proběhlo zavolání právě ukončené funkce.

Tímto jsme si detailněji popsali největší úskalí řešení implementační části projektu.

Obhajoba projektu

Na závěr se pokusme shrnout základní pravidla pro ústní prezentaci, která by vám měla pomoci při přípravě na obhajobu projektu:

- dodržet vyhrazený čas (tj. mít prezentaci dostatečně nazkoušenou, abyste věděli, jak rychle musíte mluvit, co musíte v prezentaci vynechat a vrátit se k tomu jenom v případě zájmu komise během diskuze)
- na úvod představit sebe i tým
- nesnažit se být příliš teatrální nebo nevkusně vtipný, ale také zbytečně o sobě ani svých výsledcích nepochybovat
- protože budete mít málo času, tak se nezdržujte konstatováním obecně platných faktů, které komise zná (například čtení celého zadání nebo vysvětlování principu konečného automatu). To ovšem neznamená, že tyto obecně platná fakta nemusíte znát, právě naopak!
- množství prezentačních obrazovek se doporučuje nejvýše takový jako je počet minut vyhrazených na prezentaci
- velikost fontů by měla být alespoň 18 bodů a více; místo textu v odstavcích používejte hierarchicky odsazené odrážky a detaily v případě potřeby sdělte ústně
- používejte spisovné výrazy a k programátorskému slangu se uchylujte jen v krajních či vhodných případech

- diagram či obrázek vydá za 100 slov (někdy i více), pozor však na nekontrastní barvy textu a pozadí (nejlépe když obrázek obsahuje malé množství vzájemně kontrastních barev, včetně pozadí)
- uvádějte i literaturu, ze které jste případně studovali a především přebírali obrázky (včetně citace webových stránek)

Několik tipů a rad najdete také na stránkách fakulty, které jsou sice zaměřeny na obhajoby semestrálních projektů či bakalářských prací, ale obsahují i obecně platné rady:

- http://www.fit.vutbr.cz/info/statnice/obhajoba.rp.html

Závěr

Doufáme, že vám tato příručka dopomůže k úspěšnému absolvování projektu do předmětů IFJ a IAL.

Některé odstavce mohou být mírně závislé na aktuálním zadání, proto tento dokument obsahuje v úvodu informaci o verzi zadání projektu, kterému je šit na míru. V případě připomínek, nalezených chyb či komentářů neváhejte kontaktovat některého ze hlavních autorů tohoto textu: Zbyněk Křivka (krivka@fit.vutbr.cz) nebo Roman Lukáš (lukas@fit.vutbr.cz). Budeme vděčni za každou konstruktivní kritiku nebo i nekonstruktivní chválu ☺.