Limbaje formale și tehnici de compilare

Laborator 1

Introducere

Limbajele formale și tehnicile de compilare sau translatare ale unor date dintr-un format de intrare în altul de ieșire sunt un domeniu fundamental în tehnologia informației. Calculatoarele funcționează folosind un limbaj binar, în care totul este descris doar în termeni de 1 și 0, adevărat sau fals. Totodată instrucțiunile unui microprocesor (codul mașină) sunt foarte simple, ele rezumându-se în general la transferuri de date în memorie, operații aritmetice și logice de bază, transfer de execuție condiționat sau necondiționat, etc. A programa un calculator folosind doar aceste elemente este o sarcină foarte dificilă, consumatoare de timp și care în general nu se folosește decât în anumite situații specifice. Este mult mai ușor să implementăm algoritmul dorit într-o formă mai apropiată problemei de rezolvat (implementarea sa într-un limbaj de nivel înalt) și calculatorul să convertească această implementare în instrucțiunile sale specifice (reprezentarea în cod mașină).

Cunoașterea tehnicilor de compilare aduce o serie de beneficii foarte importante, printre care enumerăm:

- Programatorul înțelege mai bine cum execută calculatorul instrucțiunile din programul scris de el, ceea ce îi permite să optimizeze mai mult codul sursă. Chiar și folosind compilatoarele actuale, între coduri scrise în feluri diferite există mari diferențe de viteză și resurse consumate. În domenii în care acest aspect este foarte important (microcontrolere, calcul de înaltă performanță, simulare, jocuri, etc.) contează fiecare optimizare care poate fi adusă unui program, iar unele dintre aceste optimizări nu pot fi înțelese cu adevărat decât dacă este știut modul în care programul sursă este translatat în cod mașină și felul în care acest cod se execută.
- Marea majoritate a aplicațiilor necesită fluxuri de intrare care sunt descrise într-un anumit format. Acestea pot fi preluate din fișiere, din rețea, etc. Formatul acestor date poate fi standard (gen XML, JSON) sau poate fi specific aplicației. Dacă formatul este standard, programatorul are la dispoziție diverse biblioteci care îl ajută să citească și să scrie date în acel format. Dacă formatul este specific aplicației, atunci programatorul trebuie să implementeze mecanismele care să-i permită să citească date, să valideze structura lor sau să le salveze. Aceste metode de citire și validare a datelor de intrare sunt de fapt obiectul tehnicilor de translatare care, dacă formatul sursă este mai complex, devin foarte importante.
- Practic aproape toate interfețele utilizator necesită validarea datelor de intrare. În acest sens programul trebuie să verifice dacă o dată/oră introdusă, un cod de identificare, mail, etc. sunt corecte. Pentru marea majoritate a acestor validări se folosesc expresii regulate, care sunt o parte integrantă a limbajelor formale, sau alte metode mai avansate, descrise tot prin intermediul limbajelor formale.
- În general programatorii trebuie să știe mai multe limbaje de programare. Cunoașterea unor aspecte fundamentale, formale ale acestor limbaje face mult mai ușoară învățarea unor limbaje noi. La fel cum și în limbajele umane obișnuite există unele aspecte de bază gen verb, substantiv sau adjectiv, care se regăsesc în toate limbile vorbite și în limbajele de programare există unele construcții fundamentale care odată știute fac procesul învățării unui nou limbaj mult mai rapid.
- Descrierea formală a unui limbaj are vaste implicații și într-adevăr putem spune că diferența fundamentală dintre un limbaj folosit de oameni în viața de toate zilele și un limbaj de programare este în principal faptul că acesta din urmă are definită pentru fiecare construcție o semantică (înțeles) foarte clară și care nu este ambiguă. În limbile vorbite există cuvinte cu mai multe înțelesuri și în același timp o construcție gramaticală poate să însemne mai multe lucruri. În limbajele de programare fiecare instrucțiune este complet definită în

toate contextele posibile, ceea ce face ca înțelesul acelei instrucțiuni să fie unic. Obișnuința de folosire a descrierilor formale aduce cu sine o mai mare claritate în gândire și în exprimare și în general formează obișnuința de a înțelege mai bine multe dintre aspectele implicate în actele de descriere și de comunicare.

In cadrul laboratorului de LFTC vom implementa un compilator simplu pentru un subset al limbajului de programare C, numit AtomC. Pentru aceasta vom parcurge etapele esențiale ale scrierii unui compilator. Aceste etape sunt:

- **Analiza lexicală** – preia caracterele individuale din codul sursă și le asamblează în atomi lexicali gen numere, identificatori, operatori, etc. Un atom lexical este componenta primară, indivizibilă care va fi folosită ulterior. Totodată unele componente care nu vor fi necesare ulterior (spații, comentarii) sunt eliminate.
- **Analiza sintactică** – asamblează atomii lexicali în construcțiile gramaticale specifice limbajului. De exemplu o expresie este formată din operanzi, operatori și paranteze dispuse conform anumitor reguli.
- **Analiza de domeniu** – asociază fiecare identificator întâlnit definiției sale. De exemplu în limbajul C o variabilă locală unei funcții este accesibilă doar în interiorul acelei funcții.
- **Analiza de tipuri** – determină pentru fiecare expresie care sunt tipurile de date folosite (de intrare) și cele de ieșire. De exemplu în C adunarea dintre un număr întreg și unul zecimal va avea ca rezultat un număr zecimal.
- **Mașina virtuală** – reprezintă limbajul de ieșire pe care-l generează compilatorul. Compilatoarele pot translata codul sursă într-un limbaj specific unui anumit microprocesor (cod mașină) sau într-un limbaj virtual, care nu corespunde unui microprocesor fizic. Java sau C# folosesc mașini virtuale.
- **Generarea de cod** – Este etapa finală în care codul sursă este translatat în instrucțiunile destinație. Totodată se pot aplica unele optimizări cum ar fi evaluarea expresiilor constante (de exemplu codul necesar pentru a calcula expresia „1+2” se substituie cu codul care folosește numărul „3”).

Rezultatul final al laboratorului va fi un compilator pentru limbajul AtomC care va fi capabil să compileze programe scrise în acest limbaj și să le execute.

Expresii regulate

Expresiile regulate sunt o modalitate de descriere a unor construcții lexicale simple. Ele sunt folosite foarte des în situațiile în care trebuie să procesăm diverse texte pentru a extrage din ele informațiile necesare sau pentru a testa validitatea lor. Practic toate limbajele de programare uzuale au suport direct pentru expresii regulate sau oferă biblioteci pentru folosirea lor. Există mai multe variațiuni în privința sintaxei expresiilor regulate și a facilităților oferite de acestea. Pentru acest laborator se va folosi un set mai restrâns, care în general se regăsește în marea majoritate a acestor formate.

O expresie regulată este comparată cu textul de intrare și rezultă **potrivire** (**match**) doar dacă acel text este conform descrierii din expresia regulată. Comparația poate fi făcută până s-a găsit prima apariție din text, sau pot fi căutate toate pozițiile din text care conțin secvențe de caractere conform expresiei date. Totodată secvențele conforme pot fi returnate pentru procesarea lor ulterioară (ex: se pot extrage toate adresele de mail dintr-un text) sau textul poate fi împărțit în subtexte având ca separatori aceste secvențe conforme (ex: un text format din mai multe linii de numere separate fiecare prin virgule poate fi împărțit prima oara după linii și apoi după virgule pentru a rezulta numerele de pe fiecare linie).

O expresie regulată este compusă în principal din:

- **litere, cifre, spațiu sau semne de punctuație care nu au o semnificație prestabilită** – aceste caractere se vor potrivi în textul de intrare exact cu ele însele. De exemplu, expresia regulată „**1024 KB**” se va potrivi (match) în textul sursă doar exact cu textul respectiv. Implicit se face distincție între litere mari și mici, deci această expresie nu se va potrivi cu „1024 Kb”. Spațiile sunt semnificative, deci textul „1024KB” nu se va potrivi.

- **. (punct)** – orice caracter
  Ex: ***a.r*** – „aer", „a0r", „a-r", „aUr"

- **clase de caractere** – semnifică o mulțime din care se alege un singur caracter. Ele încep și se termină cu paranteze drepte „**[**" „**]**" și conțin toate variantele posibile pentru caracterul dorit. Dacă se dorește testarea tuturor caracterelor dintr-un interval, capetele intervalului se scriu cu minus „**-**" între ele. Pot fi date simultan mai multe intervale și caractere individuale. Dacă o clasă de caractere începe cu căciulă „**^"** (hat), aceasta are rol de negare și se vor considera toate caracterele cu excepția celor date. În interiorul claselor de caractere operatorii își pierd semnificația și devin simple caractere: „**[a.]"** va testa caracterul „a" sau caracterul punct propriu-zis („**.**"), nu orice caracter.

  Ex: ***[abc]*** – oricare dintre caracterele „**a**",„**b**" sau „**c**", dar doar unul singur
  [0-9] – orice caracter numeric (digit)
  [^()] – orice caracter cu excepția parantezelor
  [a-zA-Z] – orice literă, mică sau mare

- ***e*₁*e*₂ (secvența de expresii regulate / succesiune)** – cele două subexpresii regulate consecutive ***e*₁** și ***e*₂** (fără niciun spațiu intre ele) se vor testa în textul sursă să existe consecutiv, fără niciun caracter intermediar. Secvența are semnificația de „**și"** (conjuncție logică).

  Ex: ***aer*** – testează existenta cuvântului „aer" (trei caractere consecutive)
  a[a-z] – orice cuvânt din doua litere mici care începe cu litera „a": „ac", „aa", „am", …

- ***e*₁|*e*₂ (alternativa)** – operatorul **sau** „**|**" testează dacă se potrivește oricare dintre expresiile date (disjuncție logică).

  Ex: ***aer|foc*** – cuvântul „aer" sau cuvântul „foc", dar nu amândouă simultan.

- **( ) (paranteze)** – schimbă ordinea de evaluare a operatorilor. Implicit operatorii unei expresii regulate se evaluează în ordinea:
  - prima oară se evaluează operatorii postfixați: *, +, ?, {}
  - se evaluează secvențele: ***e*₁*e*₂**
  - în final se evaluează alternativele: ***e*₁|*e*₂**

  Ex: ***a(e|ma)r*** – cuvântul „aer" sau cuvântul „amar". Daca am fi scris „***ae|mar***", fără paranteze, din cauză că alternativa se evaluează după secvență (alternativa are prioritate mai mică decât secvența), expresia ar fi testat cuvintele „ae" sau „mar"

- ***e*₁? (opțional)** – operatorul „**?**" face ca expresia de dinaintea sa (***e*₁**) să devină opțională (poate lipsi). Operatorul **?** este postfixat (se pune după expresie).

  Ex: ***ae?r*** – cuvântul „ar" sau cuvântul „aer". Operatorul „**?**" are o prioritate mai mare decât secvența și atunci se considera înaintea acesteia. Daca am fi scris ***(ae)?r***, s-ar fi testat cuvintele „aer" sau „r".

- ***e*₁* (opțional cu posibilitate de repetiție)** – operatorul „*" face ca expresia de dinaintea sa (***e*₁**) să se poată repeta ori de cate ori, inclusiv de 0 ori (poate lipsi). Operatorul „*" este postfixat. În literatura de specialitate, acest operator mai este denumit „operatorul stea" sau „închiderea Kleene".

  Ex: a[a-z]* - orice cuvânt care începe cu litera „a" și mai poate avea opțional oricâte alte litere mici: „a", „am", „aer", „abac". Operatorul „*" are o prioritate mai mare decât secvența și atunci se consideră înaintea acesteia. Daca am fi scris ***(a[a-z])****, s-ar fi testat secvențe opționale (inclusiv de lungime 0) formate din grupe de câte două litere mici, prima literă din fiecare grupă fiind „a": „" (secvența vidă), „am", „arab".

- ***e_1+* (obligatoriu cu posibilitate de repetiție)** – operatorul „**+**” face ca expresia de dinaintea sa (***e_1***) să se poată repeta ori de cate ori, dar este obligatoriu să apară cel puțin o dată. Operatorul „**+”** este postfixat. Expresia „***e_1+***” este echivalenta cu „***e_1e_1****”.

  Ex: -[0-9]+ - orice număr întreg negativ format din cel puțin o cifră. Operatorul „**+”** are o prioritate mai mare decât secvența și atunci se consideră înaintea acesteia. Dacă am fi scris ***(-[0-9])+***, s-ar fi testat secvențe (cel puțin o secvență), care încep fiecare cu „-”, urmat de o cifră: „-1”, „-0-5”, „-2-7-3”.

Pe lângă componentele de mai sus și pe care le vom folosi în continuare, o expresie regulată mai poate conține și alte componente, de exemplu: „**^**” - începutul șirului de intrare (sau început de linie), „**$**” – sfârșitul șirului de intrare (sau sfârșit de linie), „***e_1*{n}” – repetiție de exact „**n**” ori, „***e_1*{n,m}” – repetiție între „**n**” și „**m**” ori, etc.

Dacă un caracter are o semnificație specială în cadrul expresiilor regulate (operatori, paranteze, punct, backslash (****), etc.), acesta se poate prefixa cu „****” (backslash) pentru a i se anula semnificația specială și a-l face să conteze astfel doar ca un simplu caracter. De exemplu, pentru a căuta numărul *pi* cu 2 zecimale, va trebui să scriem o expresie de forma „**3\.14**”. Dacă avem de căutat chiar caracterul „\”, va trebui și el prefixat tot cu „\”. De exemplu „****” va căuta un singur „\”, deoarece primul „\” are doar rolul de a elimina semnificația specială a celui de-al doilea „\”. Regulile de folosire a caracterului „\” sunt astfel analogice cu folosirea sa din limbajul C, în cadrul șirurilor de caractere.

Caracterul **supra** „**/**” nu are o semnificație specială în interiorul unei expresii regulate, dar este folosit de unele programe pentru a marca începutul/sfârșitul expresiei regulate. Din acest motiv, dacă există erori la folosirea sa, acesta trebuie prefixat cu backslash „****” sau pus într-o clasă de caractere „**[\]**”.

Exemple:

- **[a-zA-Z_][a-zA-Z0-9_]*** – un identificator care începe cu orice literă (mică sau mare) sau linie de subliniere (underscore) și apoi poate continua opțional cu oricâte litere, cifre sau linii de subliniere
- **[0-9]+(\.[0-9]+)?** - un număr din cel puțin o cifră urmat opțional de o parte zecimală. Dacă partea zecimală există, ea trebuie să conțină cel puțin o cifră: „12”, „0.5”, „3.14159”
- **[a-zA-Z]+(*[.,?!\-_]? *[a-zA-Z]+)*** - Secvențe de cuvinte formate doar din litere. Se începe obligatoriu cu un cuvânt. Următoarele cuvinte sunt opționale și sunt separate între ele prin oricâte spații, urmate opțional de unul dintre caracterele „.,?!-_” și apoi iarăși opțional de oricâte spații.

Se poate constata că folosind expresii regulate se poate reprezenta o gamă variată de structuri lexicale. Structurile care nu pot fi reprezentate cu ajutorul expresiilor regulate sunt cele recursive, de exemplu folosirea corectă a parantezelor într-o expresie.

Deoarece expresiile regulate mai complexe pot fi destul de greu de înțeles, se recomandă ca acestea să fie testate pe diverse secvențe de text. Există online mai multe aplicații care realizează această testare, de exemplu http://regexpal.com/ sau https://regex101.com/. În aceste aplicații se introduce expresia regulată și un text de testat. În textul de testat vor fi evidențiate automat secvențele care corespund expresiei regulate.

Aplicația 1.1: *Într-un text există date calendaristice de forma „zi/luna/an”, în care ziua și luna sunt formate din 1-2 cifre iar anul din 2 sau 4 cifre. Să se scrie o expresie regulată care găsește toate aceste date. Nu este necesar să se facă validarea datelor calendaristice.*

Ex: Ion s-a născut pe data de 21/5/1990 și Maria pe data de 7/12/92.

Aplicația 1.2: *Propuneți un text în care există o secvență de caractere care corespunde expresiei regulate: [A-Z][a-zA-Z]*,[a-z]+ [0-9]+(\/.)*(\.|\?)*

Aplicația 1.3: *Să se scrie o expresie regulată care testează o sintaxă simplificată pentru clasele de caractere. Acestea trebuie să înceapă cu „[” și să se termine cu „]”; opțional după „[” poate fi „^”; în interior pot fi caractere individuale sau intervale de caractere; un caracter poate fi simplu sau prefixat de „\”.*

Exemple valide: [abc] [^0-9] [\r\n\t] [., !?;\-] [\\]

Exemple invalide (nu trebuie recunoscute): [\] [\\\] [a