ANALIZOR AL GRADULUI DE COMENTARE PENTRU LINIILE DE COD INTEGRABIL ÎN SOLUŢII OPEN SOURCE

Radu Bucea - Manea - Ţoniş

E-mail: radub_m@yahoo.com

Academia de Studii Economice, Bucureşti

Rezumat: Se descrie proiectarea şi implementarea unui analizor lexical, care generează documentaţia necesară optimizărilor clasice asupra codului C/C++. Analizorul conţine un modul distinct de analiză a constantelor simbolice în vederea generării statisticilor de cuvinte cheie, pe baza cărora se realizează documentaţia primară a textului sursă. Datele furnizate în acest mod servesc analizelor calitative asupra codului, în măsură să justifice oportunitatea aplicării metodelor de optimizare. Rezultatele experimentale demonstrează că programele scanate cu analizorul lexical AGCC au un timp mai redus de compilare şi generează mai puţine erori.

Cuvinte cheie: compilator, analizor lexical, constantă simbolică, optimizarea codului.

Abstract: This paper describes the design and implementation of a lexical analyzer that automatically generates documentation to assist classic code optimizations. This scanner contains one new component, a distinct module for analyzing tokens and lexemes in order to produce the basic documentation of the source code, which is not commonly found in traditional lexical analyzers. The resulted documentation accumulates profile information and supplies this information to identify new optimization opportunities that are not visible to traditional methods. Experimental results show that the AGCC lexical analyzer significantly improves the performance of C Compilers in terms of execution time and error alerts.

Key Words: Compiler, lexical analyzer, token, code optimization.

1. Introducere în teoria compilatoarelor

Un compilator este un set de programe, care traduce un text în conformitate cu sintaxa unui limbaj evoluat de programare într-un fişier obiect scris în cod maşină. Acesta este transformat de un alt program denumit *linker* în fişier executabil.

Funcţionalităţile unui compilator standard pot fi grupate în următoarele categorii de programe:

- **analizori lexicali** responsabili cu parcurgerea şi transformarea primară a textului sursă;
- **pre-procesoare** responsabile cu expandarea codului original conform specificaţiilor limbajului de macrocomenzi;
- **parsere** care analizează din punct de vedere sintactic codul intermediar şi testează compatibilitatea cu standardul limbajului de programare;
- **analizorii semantici** verifică tipul variabilelor, compară definiţia obiectelor cu referinţele lor din program şi emit avertizări;
- **generatorii de cod** transformă textul sursă validat în cod maşină după regulile limbajului de asamblare;
- **algoritmi de optimizare** care vizează programul la nivel de compilare, asamblare şi execuţie.

Operaţiile clasice de optimizare globală a codului vizează conform [2]:

0.1. **optimizarea structurilor repetitive** prin eliminarea instrucţiunilor cu operanzi a căror valoare nu se modifică de la o iteraţie la alta;

0.1. **ştergerea instrucţiunilor** al căror rezultat nu este preluat de nici o altă secvenţă de cod;

0.1. **realocarea spaţiului din memorie** alocat iniţial variabilelor frecvent utilizate în zona de regiştri.

Calitatea compilatoarelor depinde de următoarele trăsături, conform [4]:

– **calitatea codului generat**, aşa cum este definită în Standardul ISO/IEC 9126 - ca sumă a caracteristicilor de:

c_1. funcţionalitate;

c_2. fiabilitate;

c_3. ergonomicitate;

c_4. eficienţă;

c_5. mentenabilitate;

c_6. portabilitate.

- **detalierea mesajelor de eroare** în scopul identificării facile a erorilor de către dezvoltator;
- **viteza de compilare** depinde direct de calitatea şi numărul proceselor de optimizare a codului, menţionate ca opţiuni ale comenzii de compilare:
  - specificarea de biblioteci externe suplimentare;
  - generarea de cod destinat fluxurilor concurente de procesare;
  - identificarea liniilor de cod cu efecte imprevizibile;
  - generarea codului concomitent cu identificarea erorilor;
  - alocarea suplimentară de registre temporare codului compilat;
  - asigurarea unei portabilităţi maxime a codului;
  - generarea mesajelor de avertizare în cazul problemelor de compatibilitate.
- **integrarea cu instrumente de dezvoltare**, multe compilatoare fiind incluse în medii de dezvoltare ori colecţii de biblioteci software, care diminuează semnificativ viteza de compilare;
- **asigurarea de suport** online atât pentru instalarea compilatorului, cât şi pentru utilizarea sa în condiţii optime;
- **conformitatea cu Standardul C++ ISO/IEC FDIS 14882:1998;** utilizarea de extensii adăugate de un producător şi neconforme cu Standardul C++ conduce la reducerea portabilităţii codului şi măreşte dependenţa de un anumit tip de compilator; un compilator de calitate trebuie să accepte trăsături mai puţin uzuale ale Standardului C++ cum este cazul specializării parţiale a şabloanelor de funcţii şi clase;
- **preţul**, existând pe piaţă compilatoare liber download-abile şi compilatoare contra cost, după cum rezultă din Tabelul 1.

Tabelul 1. Centralizator asupra compilatoarelor comerciale şi Open source, după [4]

Compilatoare libere	Compilatoare contra cost
Apple C++	Borland C++
Bloodshed Dev-C++	CodeWarrior C++ (formerly Metrowerks)
Borland C++ v.5.5.	Comeau C++
Cygwin (GNU C++)	Edison Design Group C++ Front End
Digital Mars C++	Green Hills C++
MINGW - "Minimalist GNU for Windows"	HP C++ for Unix and HP C++ for OpenVMS
DJ Delorie's C++ development system for DOS/Windows (GNU C++)	IBM C++
GNU CC source	Intel C++ for Windows, Linux
Intel C++ for linux	Mentor Graphics/Microtec Research C++
The LLVM Compiler Infrastructure (based on GCC)	Microsoft C++
Microsoft Visual C++ 2008 Express edition	Paradigm C++
Sun Studio	The Portland Group C++
	SGI C++, optimizing compiler
	Sun C++
	WindRiver's Diab C++

Se recomandă compilatorul GNU GCC întrucât este un produs Open source, satisface majoritatea caracteristicilor de calitate şi este compilatorul standard pentru sistemul de operare Linux.

2. Cerinţe de calitate ale compilatoarelor moderne incluse în AGCC

Analizorul lexical AGCC a fost scris ca parte componentă a compilatorului GCC şi moşteneşte de la acesta principalele caracteristici, conform cu [6]:

C.1. **analiza multi-pass a codului sursă**, presupune scindarea analizorului în mai multe secvenţe de program, fiecare parcurgând o dată fişierul sursă şi operând transformările necesare; principiul descompunerii compilatoarelor în subprograme după scopul funcţional a fost fundamentat la Universitatea Carnegie Mellon în cadrul proiectului Production Quality Compiler (PQC);

C.1. **împărţirea conceptuală a analizorului în Front-End şi Back-End**, unde:

- **Front-End** construieşte tabloul de cuvinte cheie şi simboluri proprii limbajului de programare vizat, iar apoi parcurge fişierul în vederea construirii unui model intern al codului sursă; presupune următoarele etape:

  e_1. **reconstrucţia liniei**, în cazul AGCC, se realizează prin citirea caracter cu caracter şi delimitarea viitoarei linii utilizând marcatori de sfârşit de linie;

  e_2. **împărţirea codului sursă în constante simbolice** pe baza expresiilor regulate; această etapă mai poartă numele de scanare;

  e_3. **pre-procesarea** în cadrul căreia AGCC îşi construieşte un set propriu de directive care lansează în execuţie procese paralele de analiză şi transformare;

  e_4. **construirea arborelui în vederea parsării** prin înlocuirea structurii liniare dictată de constantele simbolice cu una arborescentă, conformă cu sintaxa limbajului de programare.

- **Back-End** generează codul destinaţie împărţit în codul sursă parsat şi fişierul de comentarii; generarea propriu-zisă este precedată de următoarele doua faze:

  f_1. **analiza** bazată pe reprezentarea intermediară a codului surprinde dependenţa dintre variabile; după [2], interesează colectarea datelor pentru aflarea numărului de:

  - apeluri individuale la funcţiile din program;

  - executări pentru fiecare bloc distinct de instrucţiuni;

  - validări ale condiţiei pentru fiecare ramură a unei structuri alternative şi pentru fiecare caz din structurile condiţionale de tip *CASE*.

  f_2. **optimizarea generală a codului** vizează comasarea liniilor de cod, acolo unde este posibil, şi eliminarea codului neoperaţional.

Alte cerinţe de calitate ale compilatoarelor moderne, după [5]:

- să se bazeze pe o **notaţie familiară** programatorului;
- **input minimal** necesar fluxului de prelucrări;
- **funcţionalitate transparentă** şi **predictibilă**;
- **dimensiune redusă** şi **viteză de execuţie mare**;
- **tratarea** corespunzătoare a **erorilor de parsare**;
- orientat pe **simplitate şi eficienţă**.

Analizorul AGCC satisface majoritatea cerinţelor menţionate anterior prin folosirea următoarelor tehnici, conform [1]:

- **algoritmi de sortare orientaţi pe cod**, economisind timpul necesar accesării tabelelor; eficienţa scanării depinde direct de lungimea codului sursă; următorul algoritm satisface acest criteriu urmărind frecvenţa de apariţie a caracterelor "/" şi "*":

```
if(pos[0]<=len){
        str_comp(i,0,pos[0]);
    }
else if(pos[1]<=len){
        k=true;
        k=(pos[2]<=len)?false:true;
          if(k==false){
                  str_comp(i,0,pos[1]);
                  str_comp(i,pos[2]+2,len);
          }
          else {
                  str_comp(i,0,pos[1]);
                  return;
          }
   }
else if(pos[2]<=len){
        k=false;
          str_comp(i,pos[2]+2,len);
   }
```

- **citirea în blocuri de memorie** corespunzătoare sectoarelor de disc este mai eficientă decât introducerea fiecărui caracter în parte; din acest motiv AGCC permite încărcarea întregului text sursă în memorie pentru o parcurgere mai rapidă; se consideră că viteza de execuţie a scanării creste în acest fel cu până la 30%;
- AGCC încurajează folosirea **instrucţiunilor de salt**, *goto*, întrucât nu îşi propune o analiză semantică asupra codului iar o prelucrare necesită o sigură parcurgere a textului sursă chiar daca pot rezulta mai multe transformări.

3. Ciclul de dezvoltare al analizorului lexical AGCC

În teoria calculatoarelor, analizorii lexicali convertesc secvenţe de caractere în constante simbolice. Un analizor este compus din scaner şi generatorul de constante simbolice. Specificaţia limbajelor de programare prevede seturi de reguli de sintaxă, care precizează secvenţele de caractere care pot forma constante simbolice. Constanta simbolică specifică tipul unui set de simboluri ori cuvinte cheie. Asocierea lor face obiectul funcţiilor de constante simbolice conţinute de un analizor standard.

Din punct de vedere funcţional, AGCC înlătură toate comentariile din codul sursă original şi le înlocuieşte cu caracterul SPACE. Toate caracterele BACK-SLASH sunt şterse apoi, conducând la comasarea liniilor de cod detaliate în programul original pe mai multe rânduri pentru a obţine mai multă claritate. AGCC identifică simbolurile şi cuvintele cheie, după care stabileşte tipul din care acestea fac parte. În această etapă de dezvoltare, analizorul ignoră simbolurile şi cuvintele cheie, care nu au asociate o constantă simbolică. Scanarea secvenţială a textului sursă se bazează pe tabelul de constante simbolice care în cazul AGCC se prezintă după modelul din Tabelul 2:

Tabelul 2. Cuvinte cheie şi constante simbolice

Lexeme	Tokens
//	INLINE
/*	MULTI_BEG
*/	MULTI_END
;	SEMICOL
,	COMMA
)	R_BRACKET
"="	ASSIGN
{	L_CBRACKET
:	COLON
\	BACKSLASH
#	MACRO
struct	STRUCT
class	CLASS
void	VOID
bool	BOOL
int	INT
float	FLOAT
double	DOUBLE
char	CHAR

```
<Tipuri_date>
 <Linie>
  <Nr>6</Nr>
  <Tip>class</Tip>
  <Den>Test</Den>
 </Linie>
 <Linie>
  <Nr>43</Nr>
  <Tip>int</Tip>
  <Den>testMe(int a)</Den>
 </Linie>
 <Linie>
  <Nr>51</Nr>
  <Tip>void</Tip>
  <Den>testMeToo(char c1)</Den>
 </Linie>
 <Linie>
  <Nr>57</Nr>
  <Tip>int</Tip>
  <Den>publicVar</Den>
 </Linie>
</Tipuri_date>
```

Figura 1. Structura XML de constante simbolice

Căutarea unei constante simbolice încetează o dată cu întâlnirea de spaţii sau sfârşitului de linie. După identificarea constantelor simbolice, rezultatul este parsat în format XML şi editat într-un fişier după modelul din figura 1. Pentru realizarea aplicaţiei, se utilizează mediul integrat de dezvoltare Open source Eclipse, compilatorul GCC şi colecţia de biblioteci StandardTemplateLibrary care eficientizează scrierea de cod şi menţine viteza de execuţie a programului. Codul sursă redactat conţine metode denumite sugestiv şi însoţite de comentarii clare. Executabilul obţinut după compilare este referit direct. Deoarece fazele de analiză şi optimizare pot necesita alocări suplimentare de memorie şi timpi prelungiţi de execuţie, utilizatorul le omite în modul de lansare default (fără argumente suplimentare) al programului AGCC.

Se separă şi liniile de comentarii însoţite de informaţii privind poziţia în textul sursă, obţinându-se cu ele un fişier distinct. Fişierul de comentarii este înglobat într-o structură portabilă de tip XML, care însoţeşte codul sursă sub forma unui manual de utilizare. În cazul produselor software de foarte bună calitate, dezvoltatorii includ funcţia help care se bazează pe conţinutul fişierului de comentarii, conţinut pe care îl structurează în raport cu criterii specifice aplicaţiei.

Aceste informaţii permit:

- **construirea unei liste de variabile** la care se adaugă comentarii ce decurg din textul sursă privind tipul, domeniul, frecvenţa de utilizare în cod şi legături cu alte variabile;
- **generarea comentariilor** pentru a marca semnificaţia secvenţelor de prelucrare: calcul de sume, selecţii dintr-o mulţime, construire de rapoarte şi alte operaţii cu nivel de complexitate ce depinde strict de modul în care este proiectat produsul utilitar.

4. Testarea analizorului lexical AGCC

S-a impus o testare dinamică şi analitică asupra AGCC, care presupune rularea unui program complet pentru executarea tuturor procedurilor şi scenariilor de testare aferente diverselor stiluri în redactarea şi comentarea codului C++. S-au aplicat aceleaşi proceduri de testare pentru a verifica comportamentul programului şi pe alte sisteme de operare.

Se prezintă în Tabelul 3 output-ul analizorului lexical AGCC împreună cu textul sursă:

Tabelul 3. Testarea produsului AGCC

1. Comentarea integrală a unei linii de cod cu "//":	rezultă în urma parsării cu AGCC:
int main(void){ ... // Acesta este un comentariu ... return 0; }	1: int main(void){ 2: ... 4: ... 5: return 0; 6: }
2. Comentarea parţială a unei linii de cod cu "//":	
int main(void){ ... for(int i=0;i<10;i++){ // Acesta este un comentariu ... return 0; }	1: int main(void){ 2: ... 3: for(int i=0;i<10;i++){ 4: ... 5: return 0; 6: }
3. Comentarea integrală a unei linii de cod cu "/*...*/":	
int main(void){ ... /* Acesta este un comentariu */ ... return 0; }	1: int main(void){ 2: ... 4: ... 5: return 0; 6: }
4. Comentarea parţială a unei linii de cod cu "/*...*/":	
int main(void){ ... string city[3]={"Atena", "Bucuresti", "Sofia"};/*Acesta este un comentariu */ ... return 0; }	1: int main(void){ 2: ... 3: string city[3]={"Atena", "Bucuresti", "Sofia"}; 4: ... 5: return 0; 6: }
5. Comentarea mai multor linii cu "//":	
int main(void){ ... // Acesta este un comentariu // pe mai multe // linii. ... return 0; }	1: int main(void){ 2: ... 6: ... 7: return 0; 8: }
6. Comentarea mai multor linii cu "/*...*/":	
int main(void){ ... /* Acesta este un comentariu pe mai multe linii. */ ... return 0; }	1: int main(void){ 2: ... 6: ... 7: return 0; 8: }
7. Comentarea pe mai multe linii cu "/*...*/" şi "\":	
/* */ # /* */ defi\ ne FO\ O 10\ 20 int main(void){ ... return 0; }	 2: # define FOO 1020 7: int main(void){ 8: ... 9: return 0; 10: }

Testarea s-a efectuat asupra a 53 de linii de text sursă original rezultând 38 de linii de cod necomentat după o singură parcurgere a fişierelor sursă. Au fost eliminate 15 linii de comentariu, adică 28,3% din lungimea totală, fără să fie afectate structura şi sintaxa proprie limbajului C++.

Există două excepţii de care AGCC ţine seama:

e_1. comentariile din cadrul directivelor de pre-compilare sunt ignorate;

e_2. asocierea de trigrafuri cu operatorul *backslash-new line* este evitată de AGCC prin convertirea acestora după eliminarea caracterului "\".

O practică bună în redactarea şi depanarea codului în vederea executării optime a funcţiilor AGCC şi compilării fără erori, presupune:

- verificarea **tipului parametrilor de intrare,** precum şi **valorile întoarse de funcţii**;
- **utilizarea şabloanelor** pentru eliminarea conversiilor nedorite între tipuri de date;
- **definirea de constante cu const** urmat de tipul de dată agreat;
- **declararea variabilelor** înainte de a le folosi şi **eliberarea spaţiului de memorie** alocat anterior imediat după ce au fost utilizate;
- **folosirea de paranteze** ori de cate ori logica calculului nu este foarte transparentă;
- **utilizarea assert** în tratarea excepţiilor pentru a preveni eventuale erori;
- folosirea unui **parametru de ieşire** pentru rezultat şi **return** pentru eroare când o funcţie trebuie să întoarcă o valoare calculată şi o eroare în acelaşi timp;
- **verificarea valorilor de eroare** întoarse de funcţii ale bibliotecilor de sistem pentru acele funcţii care oferă acces la resursele sistemului, cum ar fi malloc() sau open().
- o **sistematizare a erorilor posibile**, bazată pe gradul de gravitate al acestora şi pe influenţa lor diferită asupra rezultatelor finale ale programului.

Se recomandă evitarea:

- **conversiilor între tipuri de date diferite** şi, mai ales, conversia pointerilor de un tip anume de date în void*;
- **definirii de tipuri derivate din pointeri** (ex: typedef char* Sir_de_caractere) şi de constante preprocesare cu #define;
- introducerii mai multor **instrucţiuni pe aceeaşi linie** mai ales în cazul structurilor alternative şi repetitive;
- **desfăşurarea structurilor de control** pe mai mult de 3 niveluri;
- **invocarea exit()** din interiorul funcţiilor de biblioteci;
- **folosirii goto, break** sau **continue** pentru ieşirea dintr-o structură şi a structurilor repetitive post-condiţionate de tip **do{...} while()**;
- **utilizării variabilelor globale** nesustenabile în cazul fluxurilor de prelucrare concurente şi a limbajului de macrocomenzi.

5. Concluzii

Pe baza testelor desfăşurate, se impun următoarele optimizări:

- o noua **schemă de alocare a memoriei** necesare pentru execuţia AGCC;
- **documentarea comenzilor inline**;
- **rezolvarea** posibilelor **erori** sau **conflicte**;
- adăugarea în *Front-End* a facilităţilor pentru un **nou limbaj de programare**.

Se recomandă ca schimbările radicale să se producă în primele faze ale ciclului de dezvoltare astfel încât problemele care pot rezulta să fie remediate, iar adăugarea de patch-uri se va produce în urma unei analize cost-beneficiu. Se urmăreşte menţinerea complementarităţii cu standardul C++ în vigoare în cazul versiunilor următoare.

Bibliografie

1. **MÖSSENBOCK, H**.: A Generator for Production Quality Compilers. Proc. of the Third International Workshop CC '90, Schwerin, FRG, October 22-24, Springer-Verlag, 1990, pp. 42-55. ISBN 3540536698, 9783540536697.
2. **POHUA, P., CHANG, SCOTT, A., MAHLKE, WEN-MEI, W., HWU**: Using Profile Information to Assist Classic Code Optimizations, Software - Practice and Experience, Vol. 21(12), 1991, pp. 1301-1321.
3. **STROUSTRUP, B**.: The Design and Evolution of C++, Addison-Wesley, 1994. ISBN 0201543303, 9780201543308.
4. *** * *:** http://www.research.att.com/~bs/compilers.html/
5. *** * *:** http://gcc.gnu.org/onlinedocs/
6. *** * *:** http://en.wikipedia.org/wiki/