

Tema 3 - AST

- Deadline: ~~13.12.2020~~ 18.12.2020 - Temele trimise pe parcursul vacantei vor avea depunctarea asociată unei **zile de întârziere**
- Deadline HARD: 10.01.2020
- Data publicării: 30.11.2020
- **Actualizat: 12.12.2020**
- Responsabili:
 - Radu Nichita [mailto:radunichita99@gmail.com]
 - Marian Burcea [mailto:mariangabriel057@gmail.com]
 - Cristian Olaru [mailto:cristianolaru99@gmail.com]

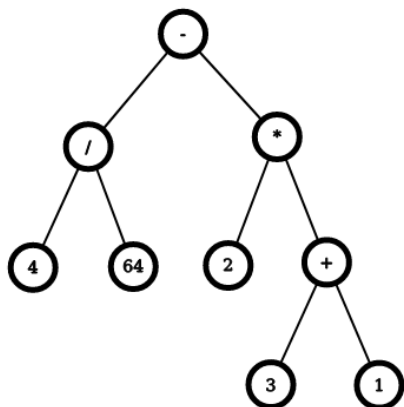
Enunț

Să se implementeze o funcție în limbaj de asamblare care efectuează parsarea unei expresii matematice în formă prefixată și construiește un AST (abstract syntax tree). Numerele ce apar în expresie sunt numere întregi cu semn, pe 32 de biți, iar operațiile ce se aplică lor sunt: +, -, /, *. Expresia prefixată va fi primită sub forma unui șir de caractere ce este dat ca parametru funcției, rezultatul fiind un pointer către nodul rădăcină al arborelui, salvat în registrul **EAX**.

Arbore sintactic abstract

Arborii sintactici abstracti sunt o structură de date cu ajutorul căreia compilatoarele reprezintă structura unui program. În urma parcurgerii AST-ului, un compilator generează metadatele necesare transformării din cod de nivel înalt în cod assembly. Puteți găsi mai multe informații despre AST aici [https://en.wikipedia.org/wiki/Abstract_syntax_tree].

Reprezentarea sub forma unui AST a unui program/expresii are avantajul de a defini clar ordinea evaluării operațiilor fără a fi necesare paranteze. Astfel expresia $4 / 64 - 2 * (3 + 1)$ poate fi reprezentată sub forma:



Implementare

Programul va folosi ca input un string în care se află parcurgerea preordine a arborelui, în ordinea, **rădăcină, stânga, dreapta**, ce poartă numele de Forma poloneză prefixată [https://en.wikipedia.org/wiki/Polish_notation]. Această expresie trebuie transformată în arbore de către funcția `create_tree(char* token)` din fișierul `ast.asm`, funcție care este apelată de checker. De asemenea, de eliberarea memoriei utilizate pentru reținerea arborelui se ocupă checkerul.

Mai mult, veți avea de implementat funcția `iocla_atoi` (în același fișier), care are o funcționalitate similară funcției `atoi` din C.

```
int iocla_atoi(char* token)
```

Se garantează că inputul primit de `iocla_atoi` este valid (un număr ce poate fi reprezentat pe 4 octeți).

Astfel, ce vă revine de făcut este să implementați cele 2 funcții (`create_tree` și `iocla_atoi`). Urmăriți comentariile din schelet pentru detalii.

De asemenea, structura folosită pentru a stoca un nod din arbore arată astfel:

```
struct __attribute__((packed)) Node
{
    char* data;
    struct Node* left;
    struct Node* right;
};
```

Nu este nevoie să definiți (sau să lucrați cu) această structură. Este important doar să știți cum este reținută în memorie.

Vă puteți folosi de funcțiile

```
int evaluate_tree(Node* root) // primește un arbore și întoarce rezultatul evaluării lui
void print_tree_inorder(Node* root) // primește un arbore și afișează nodurile în urma parcurgerii inordine.
void print_tree_preorder(Node* root) // primește un arbore și afișează nodurile în urma parcurgerii preordine.
void check_atoi(char* str) // primește un șir de caractere și afișează 'Equal' sau 'Not equal', verificând dacă iocla_atoi întoarce același rezultat ca atoi.
```

Arborele trebuie să conțină (doar && toate) simbolurile ce se găsesc în șirul primit ca input. Orice implementare care se abate de la această regulă va primi 0 pct.

Stringul data conține fie un *operator* (+, -, *, /), fie un *operand* (număr). În ambele cazuri, stringul se termină cu caracterul \0.

După cum puteți afla și de pe acest link [<https://stackoverflow.com/questions/11770451/what-is-the-meaning-of-attribute-packed-aligned4>], următorul cod:

```
__attribute__((__packed__))
```

îi interzice compilatorului să adauge padding [<https://stackoverflow.com/questions/4306186/structure-padding-and-packing>] în cadrul unei structuri, distanțele față de începutul structurii la care se află campurile acesteia fiind astfel cele așteptate și nevariind de la o mașină la alta.

Găsiți aici [https://ocw.cs.pub.ro/courses/_media/iocla/teme/check_ast.zip] un fișier schelet de la care puteți începe implementarea.

O explicație a evaluării expresiei găsiți aici [https://en.wikipedia.org/wiki/Polish_notation#Evaluation_algorithm].

Exemple de rulare

```
$ ./ast
* - 5 6 7
-7
$ ./ast
+ + * 5 3 2 * 2 3
23
$ ./ast
- * 4 + 3 2 5
15
```

Testare

Tema se poate testa pe platforma vmchecker sau local folosind checkerul check de aici [https://ocw.cs.pub.ro/courses/_media/iocla/teme/check_ast.zip].

Trimitere și notare

Temele vor trebui încărcate pe platforma vmchecker [<https://vmchecker.cs.pub.ro/ui/#IOCLA>] (în secțiunea IOCLA) și vor fi testate automat. Arhiva încărcată trebuie să fie o arhivă .zip care să conțină:

- fișierul sursă ce conține implementarea temei, denumit ast.asm
- fișier README ce conține descrierea implementării

Punctajul final acordat pe o temă este compus din:

- punctajul obținut prin testarea automată de pe vmchecker - 90%
- fișier README - 10%

A fost făcut un update al regulamentului de realizare a temelor - s-a introdus o secțiune pentru depuneri, vă rugăm să o parcurgeți. De asemenea dacă nu ați parcurs regulamentul de realizare a temelor deja vă recomandăm să o faceți.

Mașina virtuală folosită pentru testarea temelor de casă pe vmchecker este descrisă în secțiunea Mașini virtuale din pagina de resurse.

Precizări suplimentare

- Checkerul se va rula folosind comanda ./check după ce ați dat drepturi de execuție fișierului.
- Nu trebuie să afișați nimic la stdout.
- Aici [<http://ocw.cs.pub.ro/courses/iocla/bune-practici>] puteți găsi un cheatsheet, recomandări, o serie de buguri frecvente, etc.
- Expresia citită de la tastatură este validă (nu se efectuează împărțiri la 0, nu se citesc caractere diferite de [0-9] și "-+/*", etc)
- Eliberarea memoriei realizată de funcția free_ast trebuie să se execute cu succes.
- Pentru orice subarbore cu mai mult de un nod, rădăcina subarborelui este operatorul, iar fiii sunt operanzii.
- Ordinea efectuării operațiilor este **de la stânga la dreapta**.

```
$ ./ast
- 2 1
1
```

- Observați că într-un arbore sintactic abstract prioritatea operațiilor matematice este dată exclusiv de poziția acestora în cadrul arborelui. Astfel, pentru inputul: * + 2 1 + 3 4 se va afișa 21 și nu 9, operațiile executându-se în ordinea (2 + 1) * (3 + 4), și nu 2 + 1 * 3 + 4.

- Parsarea stringurilor pentru a obține numere trebuie realizată în limbaj de asamblare, **nu cu o funcție externă (cum ar fi atoi)**.

Resurse

Arhiva ce conține checkerul, testele și fișierul de la care puteți începe implementarea este aici
[https://ocw.cs.pub.ro/courses/_media/iocla/teme/check_ast.zip].

iocla/teme/tema-3.txt · Last modified: 2020/12/12 17:12 by radu.nicolau