Tema 3 - AST

- Deadline: ~~13.12.2020~~ 18.12.2020 - Temele trimise pe parcursul vacantei vor avea depunctarea asociata unei **zile de intarziere**
- Deadline HARD: 10.01.2020
- Data publicării: 30.11.2020
- **Actualizat: 12.12.2020**
- Responsabili:
  - Radu Nichita
  - Marian Burcea
  - Cristian Olaru

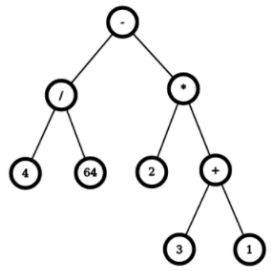
Enunț

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 primita sub forma unui șir de caractere ce este dat ca parametru funcției, rezultatul fiind un pointer către nodul rădăcină al arborelui, salvat in registrul **EAX**.

Arbore sintactic abstract

Arborii sintactici abstracți 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.

Reprezentarea sub forma unui AST a unui program/expresii are avantajul de a defini clar ordinea evaluarii operațiilor fară 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 poarta numele de Forma poloneza prefixată. Această expresie trebuie transformată în arbore de către functia `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` (in același fișier), care are o funționalitate 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 inord
void print_tree_preorder(Node* root)   // primește un arbore și afișează nodurile în urma parcurgerii preor
void check_atoi(char* str)             // primește un șir de caractere și afișează 'Equal' sau 'Not equal',
```

Arborele trebuie să conțină (doar && toate) simbolurile ce se găsesc in 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 puteti afla si de pe acest link, urmatorul cod:

```
__attribute__((__packed__)
```

- Anunțuri
- Bune practici
- Calendar
- Catalog
- Feed RSS
- IOCLA Need to Know
- Reguli și notare
- Resurse utile

Cursuri

- Curs 00: Prezentare
- Curs 01-02: Programe și sistemul de calcul
- Curs 02-03: Arhitectura sistemelor de calcul
- Curs 03: Arhitectura x86
- Curs 04: Reprezentarea datelor in sistemele de calcul
- Curs 05: Reprezentarea datelor in sistemele de calcul - C2
- Curs 06 - 07: Setul de instructiuni
- Curs 07: Declararea datelor
- Curs 08 - 09: Moduri de adresare
- Curs 09: Stiva
- Curs 10 - 11: Functii
- Curs 12: C + asm
- Curs 13: Unelte, utilitare
- Curs 13 - 15: Buffer overflows, securitate
- Curs 16 - 17: Optimizări
- Curs 18 - 19: Linking

Laboratoare

- Laborator 01: Reprezentarea numerelor, operații pe biți și lucru cu memoria
- Laborator 02: Operații cu memoria. Introducere în GDB
- Laborator 03: Toolchain
- Laborator 04: Introducere în limbajul de asamblare
- Laborator 05: Rolul registrelor, adresare directă și bazată
- Laborator 06: Lucrul cu stiva
- Laborator 07: Apeluri de funcții
- Laborator 08: Structuri, vectori. Operatii pe șiruri
- Laborator 09: Interactiunea C-assembly
- Laborator 10: Gestiunea bufferelor. Buffer overflow
- Laborator 11: Optimizări
- Laborator 12: Linking

- Laborator facultativ: ARM assembly

Teme

- Tema 1 - printf
- Tema 2 - strings
- Tema 3 - AST
- Tema 4 - Exploit ELFs

Table of Contents

îi interzice compilatorului să adauge padding în cadrul unei structuri, distanțele față de începutul structurii la care se află câmpurile acesteia fiind astfel cele așteptate și nevariind de la o mașină la alta.

Găsiți aici un fișier schelet de la care puteți începe implementarea.

O explicație a evaluării expresiei găsiți aici.

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.

Trimitere și notare

Temele vor trebui încărcate pe platforma vmchecker (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 facut un update al regulamentului de realizare a temelor - s-a introdus o secțiune pentru depunctări, vă rugăm să o parcurgeți. De asemenea daca nu ați parcurs regulamentul de realizare a temelor deja vă recomandăm sa 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 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] si "-+/* ", etc)
- Eliberarea memoriei realizata de functia `free_ast` trebuie sa 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 operatiilor matematice este dată exclusiv de poziția acestora în cadrul arborelui. Astfel, pentru inputul: `* + 2 1 + 3 4` se va afișa `21` si 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.