Laboratorul 4.
Analiză Sintactică II.
Generarea de AST.

AST - Abstract Syntax Tree

AST-ul este o structură de date construită în memorie, de către compilator, ca o rafinare a arborelui de derivare (obținut doar prin rularea parser-ului).

Această structură nu are o formă standardizată (depinde de implementarea și cerințele fiecărui compilator și limbaj de programare), dar în general are următoarele caracteristici:

- Toate nodurile, chiar dacă corespund unor tipuri diferite de instrucțiuni, sunt implementate ca fiind clase ce extind aceeași clasă de bază.
- Nu conține noduri "dummy", sau "intermediare" (ex: pentru a parsa o constantă numerică, inițial se generează un arbore de forma `expr` - `test` - `sum` - `term` - `integer`).
- Dacă unele structuri de sintaxă sunt echivalente (ex: reprezentări diferite ale constantelor numerice, `if...then...endif` vs. `if...then...else NOP endif`), ele pot fi reprezentate în AST sub o formă unificată.
- Dacă unele structuri de sintaxă există doar ca indicații pentru parser (ex: parantezele din expresiile matematice, pentru stabilirea ordinii corecte a operațiilor), nodurile lor pot fi eliminate din AST, atâta timp cât ordinea corectă a operațiilor apare corect în arbore.

Limbajul TinyC

Limbajul TinyC[1] este un alt limbaj didactic, proiectat ca fiind un subset al limbajului C.

Observați că spre deosebire de C, limbajul nu conține funcții, iar singurul tip de date folosit este al numerelor întregi. Tipurile variabilelor nu sunt specificate nicăieri în program. Considerăm ca operatorul de comparație (<) întoarce `0` dacă condiția este falsă și `1` dacă condiția este adevărată. Operatorul de atribuire (=) întoarce valoarea la care se evaluează expresia din dreapta (regula aceasta ne permite să scriem atribuiri de forma `a = b = 0`);

Specificația completă a gramaticii este disponibilă în continuare:

[1] Bazat pe cel de aici: `http://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c`

```
<program> ::= <statement>+
<statement> ::= "if" <paren_expr> <statement> |
                "if" <paren_expr> <statement> "else" <statement> |
                "while" <paren_expr> <statement> |
                "do" <statement> "while" <paren_expr> ";" |
                "{"  <statement>*  "}" |
                <expr> ";" |
                ";"
<paren_expr> ::= "(" <expr> ")"
<expr> ::= <test> | <id> "=" <expr>
<test> ::= <sum> | <sum> "<" <sum>
<sum> ::= <term> | <term> "+" <sum> | <term> "-" <sum>
<term> ::= <id> | <int> | <paren_expr>
<int> ::= <hex_int> | <bin_int> | <dec_int>
<hex_int> ::= "0" ("x" | "X") ([0-9]|[a-f]|[A-F])+
<bin_int> ::= "0" ("b" | "B") ("0"|"1")+
<dec_int> ::= [0-9]+
<id> ::= [a-z]+
```

Exerciții:

Definirea ierarhiei de clase pentru AST

Definiți ierarhia de clase. Porniți de la clasa-părinte (`ASTBaseNode`), după ca extindeți cu clase specializate (pentru blocuri de instrucțiuni, pentru `if`).

Indicații:

- Pentru blocurile de instrucțiuni (delimitate cu `{, }`) puteți pune câmpuri de forma `LinkedList<ASTBaseNode>`.
- Modelați clasa corespunzătoare instrucțiunilor `if` astfel încât să conțina 3 noduri: condiția, ramura `then` și ramura `else`.
- Modelați clasa corespunzătoare constantelor numerice astfel încât orice reprezentare a unui număr gasim în program (binar, zecimal, hexazecimal) să fie asociată unei singure clase (care conține valoarea numerică).

Implementați pentru fiecare clasă o metodă de afișare a conținutului (exemplu de de antet: `printContents(int indentation_level)`). Aceasta va face o parcurgere în adâncime a arborelui și o afișare în forma "prefix" (pe primele linii se afișează informații despre tipul și conținutul nodului-părinte, și apoi, recursiv, pe un alt nivel de indentare, și subarborii acestuia).

Generarea AST-ului și afișarea sa

Folosiți implementarea de **visitors** pentru a genera AST-ul (instanțiat și inițializat clasele definite anterior).

Afișați nodul-părinte al AST-ului, care recursiv va declanșa afișarea întreg arborelui.