CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Syllabus

- Posted on website
- Academic Integrity
Textbook

Classic text.

You should hang on to this one.
Team formation

- If you have a team, please list members in response to Piazza post.
picking up where we left off...
Deep understanding - ex 2

\[ f() + g() \times h(); \]

What is the order of the function calls?

Must \( g \) be called before \( f \)?
Deep understanding - ex 2

\[ f() + f() \times f(); \]

How many times will \( f \) be called?

Could it be just once?

If it cannot be just once, is order important?
Deep understanding - ex 2

\[ f() + f() \times f(); \]

If the value of \( f() \) depends on mutable persistent state, then the value returned by each call can be different.
Deep understanding - ex 2

\[ f() + f() \times f(); \]

If \( f \) is known to be referentially transparent, then each call to \( f() \) will produce the same value. We can then compute \( f \) once, and use its value multiple times.
Referential transparency and referential opacity are properties of parts of computer programs. An expression is called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior. This requires that the expression be pure, that is to say the expression value must be the same for the same inputs and its evaluation must have no side effects. An expression that is not referentially transparent is called referentially opaque.


If f is known to be referentially transparent, then each call to f() will produce the same value. We can then compute f once, and use its value multiple times.
What determines program meaning?

```c
#include <stdio.h>

int main() {
    int i = 0;
    int sum = 0;
    while (i <= 10) {
        sum = sum + i;
        printf("sum of integers from 0 to %d is %d.\n", i, sum);
        i = i + 1;
    }
}
```
What determines program semantics?

```c
#include <stdio.h>

int main() {
    int i = 0;
    int sum = 0;
    while (i <= 10) {
        sum = sum + i;
        printf("sum of integers from 0 to %d is %d.\n",i,sum);
        i = i + 1;
    }
}
```
What is this?

```c
#include <stdio.h>

int main() {
    int i = 0;
    int sum = 0;
    while (i <= 10) {
        sum = sum + i;
        printf("sum of integers from 0 to %d is %d.\n",i,sum);
        i = i + 1;
    }
}
```
What is this?

#include <stdio.h>

int main() {
    int i = 0;
    int sum = 0;
    while (i <= 10) {
        sum = sum + i;
        printf("sum of integers from 0 to %d is %d.\n",i,sum);
        i = i + 1;
    }
}
La suite de Syracuse est définie ainsi :
- on part d'un entier ;
- s'il est pair, on le divise par 2 ;
- sinon, on le multiplie par 3 et on ajoute 1 ;
- on recommence la même opération sur l'entier obtenu, et ainsi de suite ;
- la suite s'arrête si on arrive à 1.

syracuse :
  durée est un nombre
  e est un nombre
  début
    e prend 14
    tant que e != 1 lis
      durée prend durée + 1
      si (e mod 2) = 0, e prend e / 2
      sinon e prend e * 3 + 1
    affiche e
  ferme
  affiche "durée = {durée}"
/* The Syracuse sequence is defined as follows:  
- it starts with any natural number > 0  
- if it is even, we divide by 2  
- else we multiply by 3 and add 1  
- the process is repeated on the result  
- the process ends when the result is 1 */

void syracuse() {
    int iterations;
    int e;

    iterations = 0;
    e = 14;
    while (e != 1) {
        iterations = iterations + 1;
        if ((e % 2) == 0) e = e / 2;
        else e = e * 3 + 1;
        printf("%d\n", e);
    }
    printf("iterations = %d\n", iterations);
}
syracuse :
  durée est un nombre
  e est un nombre
  début
    e prend 14
    tant que e != 1 lis
      durée prend durée + 1
    si (e mod 2) = 0, e prend e / 2
    sinon e prend e * 3 + 1
  affiche e
  ferme
  affiche "durée = {durée}"
syracuse :

durée est un nombre
e est un nombre
début

   e prend 14
tant que e != 1 lis
durée prend durée + 1
   si (e mod 2) = 0, e prend e / 2
   sinon e prend e * 3 + 1
affiche e
ferme
affiche "durée = {durée}"
Syntax and semantics

- Syntax: program structure
- Semantics: program meaning
- Semantics are determined (in part) by program structure.
Languages: the Chomsky hierarchy

"On Certain Formal Properties of Grammars" published 1959

- recursively enumerable
- context-sensitive
- context-free
- regular

https://upload.wikimedia.org/wikipedia/commons/8/86/Noam_chomsky.jpg
grammars (generators) and languages

automata (acceptors)

recursively enumerable language

turing machine

context-sensitive language

linear-bounded automaton

context-free language

push-down automaton

regular

finite-state

language

automaton

the traditional Chomsky hierarchy


LICENSE: http://creativecommons.org/licenses/by/3.0/
Phases of a compiler

Figure 1.6, page 5 of text
Phases of a compiler

Figure 1.6, page 5 of text
Lexical Structure

```c
int main()
```

Lexical Structure

int main()
{

c
}
Lexical Structure

character stream -> token stream

int main()

int main() { id("int") id("main") LPAR RPAR LBRACE
Lexical Structure

tokens

- keywords (e.g. static, for, while, struct)
- operators (e.g. <, >, <=, =, ==, +, -, &, .)
- identifiers (e.g. foo, bar, sum, mystery)
- literals (e.g. -17, 34.52E-45, true, 'e', "Serenity")
- punctuation (e.g. {, }, , (, ), , ;)
Describing lexical structure

- We need some formal way of describing the lexical structure of a language.
meta vs object language

- **object language**: the language we are describing
- **meta language**: the language we use to describe the object language
meta vs object language

- How do we distinguish between the two?
meta vs object language

- use quotes (meta vs ‘object’)
- punctuation (e.g. ‘{’, ‘}', ‘(’, ‘)’, ‘,’ ‘;’)

- use font or font property (meta vs object)
- punctuation (e.g. `{`, `}`, `(`, `)`, `,`, `;`
Formally, a language is a set of strings over some alphabet.

- Ex. \{00, 01, 10, 11\} is the set of all strings of length 2 over the alphabet \{0, 1\}.
- Ex. \{00, 11\} is the set of all even parity strings of length 2 over the alphabet \{0, 1\}.
Formally, a grammar is defined by 4 items:

1. $N$, a set of non-terminals
2. $\Sigma$, a set of terminals
3. $P$, a set of productions
4. $S$, a start symbol

$G = (N, \Sigma, P, S)$
languages & grammars

N, a set of non-terminals
Σ, a set of terminals (alphabet)

\[ N \cap \Sigma = \{\} \]

P, a set of productions of the form (right linear)

\[ X \rightarrow a \]
\[ X \rightarrow aY \]
\[ X \rightarrow \varepsilon \]

\[ X \in N, Y \in N, a \in \Sigma, \varepsilon \] denotes the empty string

S, a start symbol

\[ S \in N \]
languages & grammars

N, a set of non-terminals
Σ, a set of terminals (alphabet)
\[ N \cap \Sigma = \{\} \]
P, a set of productions of the form (right linear)

In computer science, a linear grammar is a context-free grammar that has at most one nonterminal in the right hand side of each of its productions.

Two special types of linear grammars are the following:
- the left-linear or left regular grammars, in which all nonterminals in right hand sides are at the left ends;
- the right-linear or right regular grammars, in which all nonterminals in right hand sides are at the right ends.

https://en.wikipedia.org/wiki/Linear_grammar
Given a string $\alpha A$, where $\alpha \in \Sigma^*$ and $A \in N$, and a production $A \rightarrow \beta \in P$
we write $\alpha A \Rightarrow \alpha \beta$ to indicate that $\alpha A$ derives $\alpha \beta$ in one step.

$\Rightarrow^k$ and $\Rightarrow^*$ can be used to indicate $k$ or arbitrarily many derivation steps, respectively.