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

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?

```
#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?

#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;
    }
}
```
/*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}"
Keywords have no inherent meaning.
Program meaning is given by formal semantics.
Compiler must preserve semantics of source program in translation to low level form.
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-bound automaton

context-free language

push-down automaton

regular finite-state language

finite-state automaton

the traditional Chomsky hierarchy

Syntactic structure

Lexical structure
Phases of a compiler

Figure 1.6, page 5 of text
Phases of a compiler

Figure 1.6, page 5 of text
Lexical Structure

int main(){
Lexical Structure

```c
int main()
{
    character stream

    int main()
```
Lexical Structure

character stream $\rightarrow$ token stream

```c
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
How do we distinguish between the two?

meta vs object language
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

$G = (N, \Sigma, P, S)$
Lexical analysis: a bird's eye view

Language: a set of strings

\[ \{ \text{for, while, x, factorial, ...} \} \]

Grammar: rules for generating language

\[ G = (N, \Sigma, P, S) \]

Regular expression

Regex: a form of grammar

Finite automaton

A machine for language

C program

Generated by FLEX
languages & grammars

$N$, a set of non-terminals

$\Sigma$, 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

$S \in N$
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
Formally, a grammar $G = (N, \Sigma, P, S)$ is defined by 4 items:
1. $N$, a set of non-terminals
2. $\Sigma$, a set of terminals (alphabet)
   $N \cap \Sigma = \{\}$ ← general grammar constraints (in blue)
3. $P$, a set of productions of the form (right linear)
   $X \rightarrow a$ ← right linear grammars describe
   $X \rightarrow aY$ regular languages (in green)
   $X \rightarrow \varepsilon$
   $X \in N, Y \in N, a \in \Sigma, \varepsilon$ denotes the empty string
4. $S$, a start symbol
   $S \in N$
Given a string αA, where α ∈ Σ* and A ∈ N, and a production
A → β ∈ P
we write αA => αβ to indicate that αA derives αβ in one step.

=>^k and =>^* can be used to indicate k or arbitrarily many derivation steps, respectively.
$L(G)$ is the set of all strings derivable from $G$ starting with the start symbol; i.e. it denotes the language of $G$. 

languages & grammars
Given a grammar $G$ the language it generates, $L(G)$, is unique.

Given a language $L$ there are many grammars $H$ such that $L(H) = L$. 

languages & grammars
Given a grammar $G$ the language it generates, $\mathcal{L}(G)$, is unique. Given a language $L$ there are many grammars $H$ such that $\mathcal{L}(H) = L$.

Think about what this means for us: there is no single "correct" grammar for a language.

In fact, grammars for users vs. tool writers vs. compiler writers can all be different.

Given a language $L$ there are many grammars $H$ such that $\mathcal{L}(H) = L$. 
Lexical Analysis

- Lexical structure described by regular grammar
- Deterministic finite state machine performs analysis
How is a regular language defined?

- Recall that a language is a set of strings. This set can be finite or infinite.

- The possible regular languages over a given alphabet are defined inductively - construction given on next two slides.
LANGUAGE operations

base cases

- \{ \varepsilon \} is a regular language
- \forall a \in \Sigma, \{ a \} is a regular language

Recall, \varepsilon is the empty string
LANGUAGE operations

If \( L \) and \( M \) are regular, so are:

- \( L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \) \text{ Union}
- \( LM = \{ st \mid s \in L \text{ and } t \in M \} \) \text{ Concatenation}
- \( L^* = \bigcup_{i=0,\infty} L^i \) \text{ Kleene closure}

No other languages are regular

\( L^i \) is \( L \) concatenated with itself \( i \) times:
- \( L^0 = \{ \varepsilon \} \), by definition
- \( L^1 = L \)
- \( L^2 = LL \)
- \( L^3 = LLL \), etc.
- \( L^* \) is the union of all these sets!
Example of $L^*$

Suppose $L$ is $\{a, \text{bb}\}$

$L^0 = \{\varepsilon\}$, by definition

$L^1 = L = \{a, \text{bb}\}$

$L^2 = LL = \{\text{aa, abb, bba, bbbb}\}$

$L^3 = LLL = \{\text{aaa, aabb, abba, abbbbb, bbaa, bbbba, bbba, bbabbb, bbbba, bbbbbb, abbb, bbabb}\}$

$L^4 = \ldots \text{and so so...}$

$L^* = \bigcup_{i=0}^{\infty} L^i = \{\varepsilon, a, \text{bb, aa, abb, bba, bbbb, aab, abba, abbbbb, bbba, bbaa, bbbba, bbbba, bbbbb, abbb, bbabb, bbba, bbbbbb, abbb, bbabb, \ldots}\}$
Some regular languages over $\Sigma = \{0,1\}$

The base cases yield these regular languages:

$\{\varepsilon\}, \{0\}, \{1\}$

The inductive cases yield many more. Some are:

$\{0, 1\}, \{01\}, \{10\}, \{01, 10\}, \{0, 01\}, \{1, 01\}, \{0, 10\}, \{1, 10\}, \{0, 1, 01\}, \{0, 1, 10\}, \{0, 01, 10\}, \{1, 01, 10\}, \{00\}, \{000\}, \{0000\}, \{11\}, \{111\}, \{1111\}, \text{and many many more.}$

Can you demonstrate how each of these is regular?
Why use grammars?

- Recall that a language is a possibly infinite set of strings.
- A grammar gives us a way to describe, using finite means, an infinite set.
- Regular expressions are equivalent to regular grammars in expressive power: both regular grammars and regular expressions describe regular languages.
Inductive definition of REGular EXpressions (regex) over a given alphabet $\Sigma$

- $\varepsilon$ is a regex
- $L(\varepsilon) = \{\varepsilon\}$

For each $a \in \Sigma$, $a$ is a regex

$L(a) = \{a\}$
Assume $r$ and $s$ are regexes.

$r|s$ is a regex denoting $L(r) \cup L(s)$
$rs$ is a regex denoting $L(r)L(s)$
$r^*$ is a regex denoting $(L(r))^*$
$(r)$ is a regex denoting $L(r)$

**Precedence:** Kleene closure $> \text{ concatenation} > \text{ union}$

**Associativity:** all left-associative (minimize use of parentheses: $(r|s)|t = r|s|t$)
Algebraic laws

Assume \( r \) and \( s \) are regexes.

**Commutativity** \( r|s = s|r \)

**Associativity** \( r|(s|t) = (r|s)|t \) and \( r(st) = (rs)t \)

**Distributivity** \( r(s|t) = rs|rt \) and \( (s|t)r = sr|tr \)

**Identity** \( \varepsilon r = r \varepsilon = r \)

**Idempotency** \( r** = r^* \)
We can describe a regular language using a regular expression.
Why do we care?

- We will be using a tool called FLEX to construct a lexical analyzer (a lexer) for the programming language we're constructing a compiler for.
- If we give FLEX a regular expression describing the lexical structure of our language, FLEX will produce a C program which acts as our lexer.
- The next step for us to understand (at a high level) how FLEX converts a regex to a C program.