CSE443
Compilers

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

www.cse.buffalo.edu/faculty/alphonce/SP18/CSE443
piazza.com/buffalo/spring2018/cse443
BUILD
A
COMPILER!
Assessment plan

- Homework - 5 assignments
- Project - 5 phases / checkpoints
- Examination - 3 hour final, based on homework/project
## Learning outcomes

<table>
<thead>
<tr>
<th>Learning outcome</th>
<th>Instructional methods</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify and describe the function of the major phases of a compiler.</td>
<td>Lecture-based instruction Hands-on activities in lecture and recitation</td>
<td></td>
</tr>
<tr>
<td>Define formally the grammars used in the front end of a compiler, their application in the front end, and techniques for parsing such grammars.</td>
<td></td>
<td>HW, EX</td>
</tr>
<tr>
<td>Evaluate (compare and contrast) different intermediate representations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explain the compiler’s role in creating and managing run-time environments.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explain and evaluate (compare and contrast) different approaches to code generation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identify and explain the applicability and operation of code optimizations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build both the front and back ends of a compiler.</td>
<td></td>
<td>PROJ</td>
</tr>
</tbody>
</table>
# Grading

<table>
<thead>
<tr>
<th>INDIVIDUAL</th>
<th>TEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homework</td>
<td>Project</td>
</tr>
<tr>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>Exam</td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
</tr>
</tbody>
</table>
Teams & Recitations

- Form teams this week - I recommend teams of size 3.

- Recitations start this week.

- Recommend all team members attend same recitation, but not required (you can attend either recitation).

- For project, you may choose either C or SML - decide with your teammates (you must have a unanimous decision). We will discuss more in recitation this week.
Goal: build a compiler

source program

executable
Phases of a compiler produce an executable program from source code.

1. **Symbol Table**
2. **Machine-Dependent Code Generator** generates machine code.
3. **Intermediate representation** is processed to produce machine code.
4. **Machine-Independent Code Generator** produces intermediate code.
5. **Syntax Analyzer** processes the token stream.
6. **Semantic Analyzer** processes the syntax tree.
7. **Lexical Analyzer** processes the character stream.

Figure 1.6, page 5 of text.
Why?

- Deeper understanding of languages
- Become a better programmer
- Learn how to build tools
- Build special-purpose languages (DSLs)
- Theory meets practice
- High-level meets low-level

=> CSE490 Computer Architecture collaboration
Deep understanding - ex 1

name

vs

identifier

vs

variable
name

y.x

identifier

refers to

variable location in memory
void foo() {
    int x = 0;
    printf(x);
}

void bar() {
    double x = 3.8;
    printf(x);
}
Deep understanding - ex 1

```c
int func(int x) {
    if (x == 0) { return 1; }
    else { return x * func(x-1); }
}
```
Deep understanding - ex 1

```c
struct Pair {
    int x;
    int y;
};

void bar() {
    Pair r, s;
}
```
variables in distinct scopes, variables in distinct records/objects, or variables in distinct function invocations
Does source code completely determine order of evaluation/
execution at machine language level?
Deep understanding - ex 2

\[ a + b \times c; \]

What is the order of evaluation?
Deep understanding - ex 2

What is the order of evaluation?

\( f(x) + g(x) \times h(x); \)
Deep understanding - ex 2

\[ f(0) + f(0) \times f(0); \]

What is the order of evaluation?
Deep understanding – ex 2

\[ a + b \times c; \]

In most languages the result will be consistent with the evaluation of\[ a + (b \times c) \]
Deep understanding - ex 2

\[ a + b \times c; \]

Order of operations is important here, but order of evaluation of the variables a, b, and c is not (as long as they are evaluated before they are needed.)
Deep understanding - ex 2

\[ f(0) + g(0) \times h(0); \]

What is the order of the function calls?

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

\[ f(0) + f(0) \times f(0); \]

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.
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?

```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}"

void syracuse() {
    int iterations = 0;
    int e;
    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);
}
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-boundary

automaton

context-free language

push-down

automaton

regular finite-state

language

automaton

the traditional Chomsky hierarchy

**Lexical structure**

**Syntactic structure**
Phases of a compiler

Figure 1.6, page 5 of text