COS320: Compiling Techniques

Instructor: Zak Kincaid
TA: Vivienne Goyal

January 29, 2026

What is a compiler?

- A **compiler** is a program that takes a program written in a *source language* and translates it into a functionally equivalent program in a *target language*.
  - `gcc` : C → x86/ARM assembly
  - `javac` : Java → Java bytecode
  - `cfront` : C++ → C
  - ....

What is a compiler?

- A **compiler** is a program that takes a program written in a *source language* and translates it into a functionally equivalent program in a *target language*.
  - `gcc` : C → x86/ARM assembly
  - `javac` : Java → Java bytecode
  - `cfront` : C++ → C
  - ....

- A compiler can also:
  - Report errors & potential problems
    - Uninitialized variables, type errors, ...
  - Improve ("optimize") the program

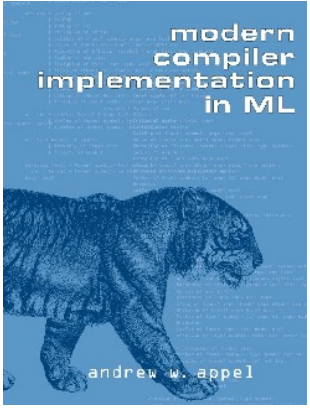
Why take COS320?

You will learn:

- **How high-level languages are translated to machine language**
- How to be a better programmer
  - What can a compiler do?
  - What can a compiler *not* do?
- Lexing & Parsing
- (Some) functional programming in OCaml
- A bit of programming language theory
- A bit of computer architecture

Course resources

- **Website**: `http://www.cs.princeton.edu/courses/archive/spr26/cos320/`
  - Assignments available through **canvas**
  - Discussion forum on **ed**
- **Office hours**:
  - Monday 3:30–5:00pm (Zak)
  - Wednesday 10–11am (Vivienne)
  - Friday 10–11am (Vivienne)
  - or by appointment
- Recommended textbook:
  Modern compiler implementation in ML (Appel)
- Real World OCaml (Minsky, Madhavapeddy, Hickey)
  `realworldocaml.org`



Grading

Homework teaches the practice of building a compiler; midterm & final skew towards theory.

- 60% Homework
  - 5 assignments, not evenly weighted
  - Expect homework to be time consuming!
- 20% Midterm
  - Thursday March 5, in class
- 20% Final

Homework policies

- Homework can be done individually or in pairs
- Due on Mondays at 11pm, with 1 hour grace period
- Can be submitted max 4 days late. 10% penalty per day late, with first four late days (across all assignments) waived.
- Feel free to discuss with others or LLMs at **conceptual** level.
  Submitted work should be your own.

Compilers

(Programming) language = syntax + semantics

- **Syntax**: what sequences of characters are valid programs?
  - Typically specified by context-free grammar

```
<expr> ::= <integer>
         | <variable>
         | <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
```

- **Semantics**: what is the behavior of a valid program?
  - *Operational semantics*: how can we execute a program?
    - In essence: an interpreter
  - *Axiomatic semantics*: what can we prove about a program?
  - *Denotational semantics*: what mathematical function does the program compute?

(Programming) language = syntax + semantics

- **Syntax**: what sequences of characters are valid programs?
  - Typically specified by context-free grammar

```
<expr> ::=<integer>
         |<variable>
         |<expr> + <expr>
         |<expr> * <expr>
         |(<expr>)
```

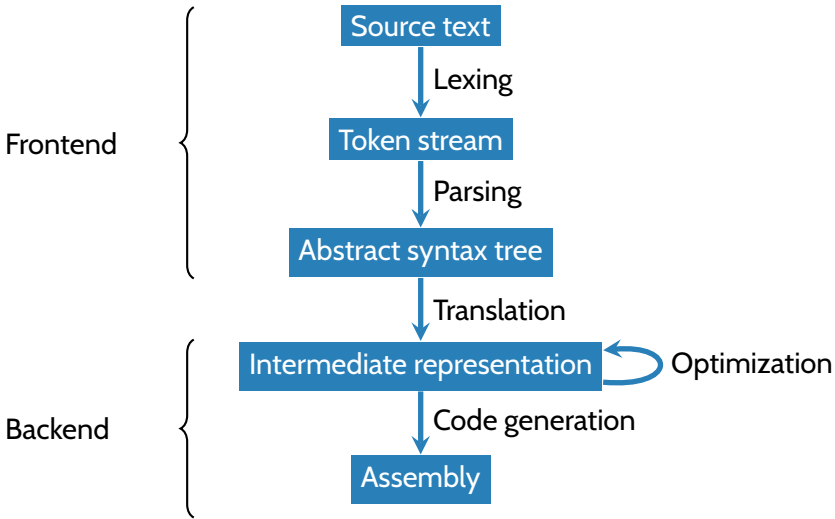
- **Semantics**: what is the behavior of a valid program?
  - *Operational semantics*: how can we execute a program?
    - In essence: an interpreter
  - *Axiomatic semantics*: what can we prove about a program?
  - *Denotational semantics*: what mathematical function does the program compute?

The job of a compiler is to translate from the syntax of one language to another, but preserve the semantics.

```c
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

```
1   factorial:
2       movl    $1, %rax
3       cmpq    $2, %rdi
4       jl      .LBB0_2
5   .LBB0_1:
6       imulq   %rdi, %rax
7       decq    %rdi
8       cmpq    $1, %rdi
9       jg      .LBB0_1
10  .LBB0_2:
11      retq

13  main:
14      movl    $.str, %rdi
15      movl    $720, %rsi
16      callq   printf
17      retq

19  .globl  .str
20  .str:
21      .asciz  "Factorial is %ld\n"
22
```

Compiler phases (simplified)

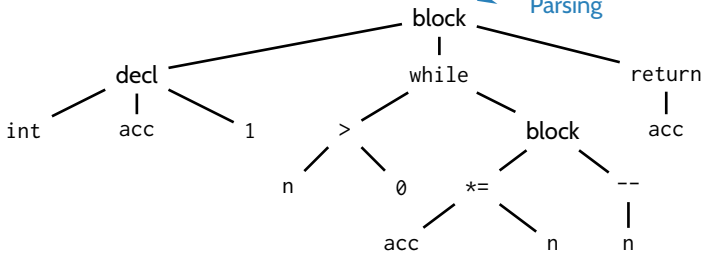


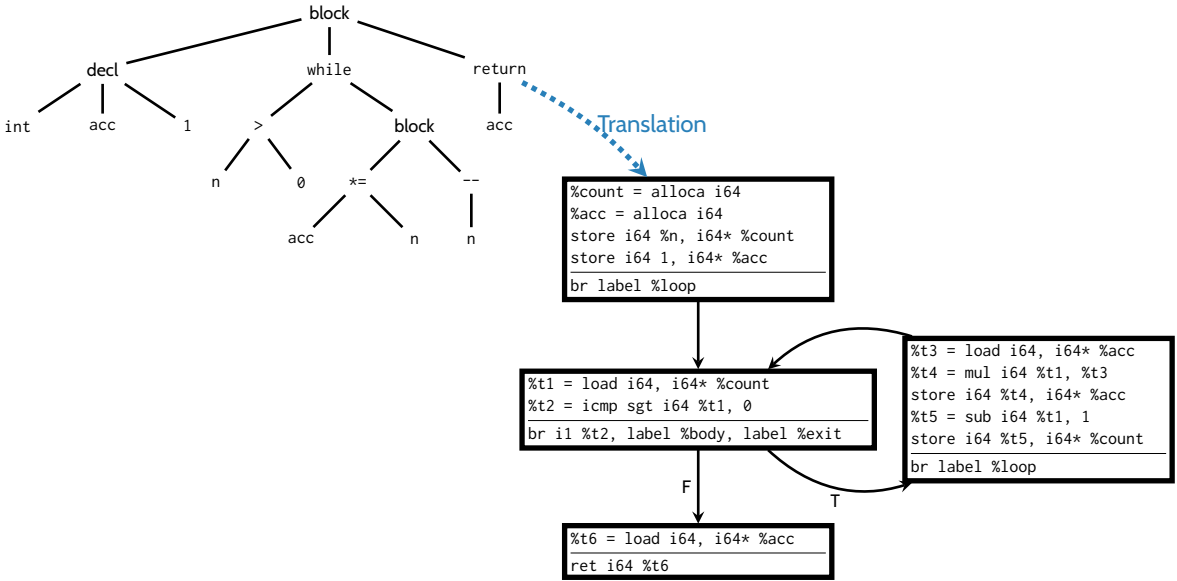
```
1  int acc = 1;
2  while (n > 0) {
3    acc *= n;
4    n --;
5  }
6  return acc;
7
```

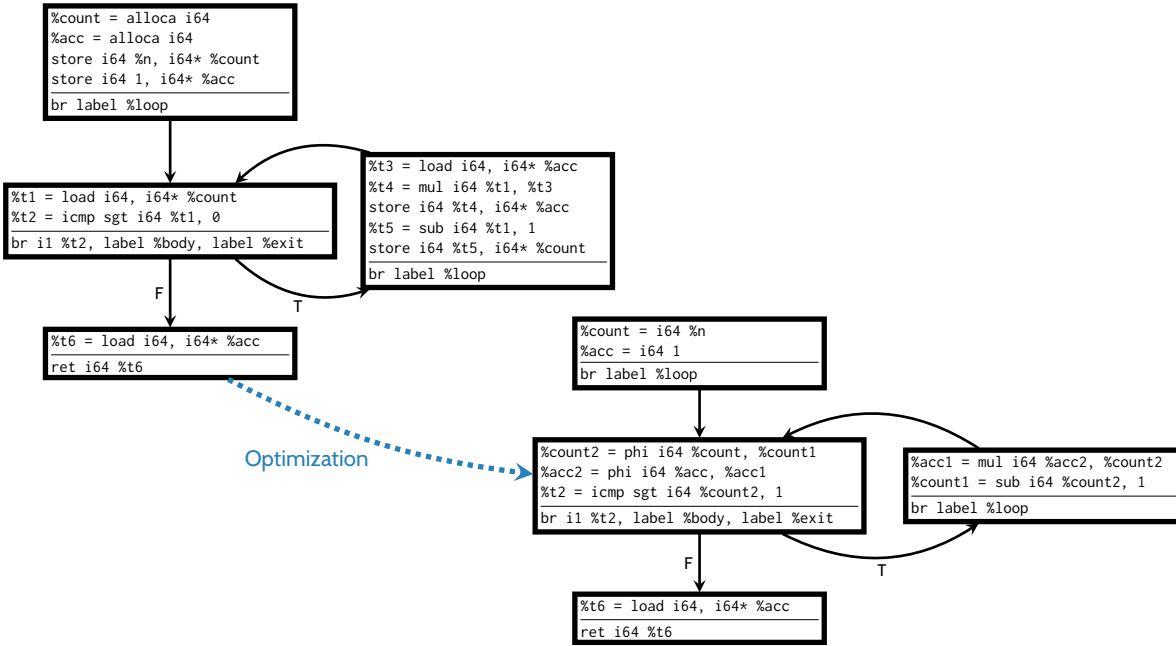
Lexing

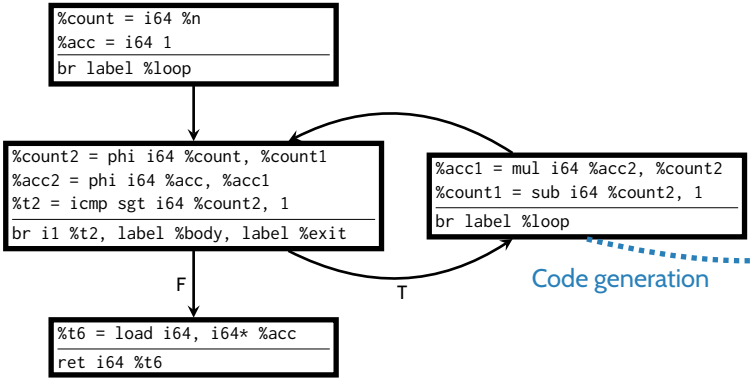
```
1  INT, IDENT "acc", EQUAL, INT 1, SEMI,
2  WHILE, LPAREN, IDENT "n", GT, INT 0, RPAREN, LBRACE,
3  IDENT "acc", TIMESEQUAL, IDENT "n", SEMI,
4  IDENT "n", DECREMENT, SEMI,
5  RBRACE
6  RETURN, IDENT "acc", SEMI
```

Parsing




block
decl
while
return
int
acc
1
>
block
acc
n
0
*=
--
acc
n
n
Translation
%count = alloca i64
%acc = alloca i64
store i64 %n, i64* %count
store i64 1, i64* %acc
br label %loop
%t1 = load i64, i64* %count
%t2 = icmp sgt i64 %t1, 0
br i1 %t2, label %body, label %exit
%t3 = load i64, i64* %acc
%t4 = mul i64 %t1, %t3
store i64 %t4, i64* %acc
%t5 = sub i64 %t1, 1
store i64 %t5, i64* %count
br label %loop
F
T
%t6 = load i64, i64* %acc
ret i64 %t6


%count = alloca i64
%acc = alloca i64
store i64 %n, i64* %count
store i64 1, i64* %acc
br label %loop
%t1 = load i64, i64* %count
%t2 = icmp sgt i64 %t1, 0
br i1 %t2, label %body, label %exit
F
T
%t3 = load i64, i64* %acc
%t4 = mul i64 %t1, %t3
store i64 %t4, i64* %acc
%t5 = sub i64 %t1, 1
store i64 %t5, i64* %count
br label %loop
%t6 = load i64, i64* %acc
ret i64 %t6
Optimization
%count = i64 %n
%acc = i64 1
br label %loop
%count2 = phi i64 %count, %count1
%acc2 = phi i64 %acc, %acc1
%t2 = icmp sgt i64 %count2, 1
br i1 %t2, label %body, label %exit
%acc1 = mul i64 %acc2, %count2
%count1 = sub i64 %count2, 1
br label %loop
F
T
%t6 = load i64, i64* %acc
ret i64 %t6




```
1   factorial:
2     movl   $1,%rax
3     cmpq   $2,%rdi
4     jl     .LBB0_2
5   .LBB0_1:
6     imulq  %rdi,%rax
7     decq   %rdi
8     cmpq   $1,%rdi
9     jg     .LBB0_1
10  .LBB0_2:
11    retq
12
```

COS320 assignments

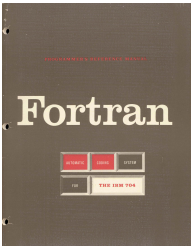
By the end of the course, you will build (in OCaml) a complete compiler from a high-level type-safe language ("Oat") to a subset of x86 assembly.

- HW1: X86lite interpreter
- HW2: LLVMlite-to-X86lite code generation
- HW3: Lexing, Parsing, Oat-to-LLVMlite translation
- HW4: Higher-level features
- HW5: Analysis and Optimizations

We will use the assignments from Penn's CIS 341, provided by Steve Zdancevic.

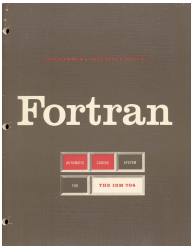
Historical note

- First "modern" compiler for FORTRAN developed at IBM in 1957
  - Grace Hopper's 1951 A-0 loader/linker
- 18 person-years to complete
- Led by John Backus, who won 1977 Turing award



Historical note

- First "modern" compiler for FORTRAN developed at IBM in 1957
  - Grace Hopper's 1951 A-0 loader/linker
- 18 person-years to complete
- Led by John Backus, who won 1977 Turing award
- You will implement one in a semester



OCaml


modern
compiler
implementation
in Java
andrew w. appel
modern
in C
. appel
modern
in ML
. appel

- Why OCaml?
  - Algebraic data types + pattern matching are *very* convenient features for writing compilers
- OCaml is a *functional* programming language
  - *Imperative* languages operate by mutating data
  - *Functional* languages operate by producing new data
- OCaml is a *typed* language
  - Contracts on the values produced and consumed by each expression
  - Types are (for the most part) *automatically inferred*.
    - Good style to write types for top-level definitions

- We recommend using VSCode + Docker for OCaml development
  - Each assignment comes with a dev container to make this simple
  - See “Toolchain” instructions on the HW page to get started
- If you have difficulty with installation, ask on ed

Intro to Formal Language Theory

Formal Languages

- An *alphabet* Σ is a finite set of symbols (e.g., $\{0, 1\}$, ASCII, unicode, tokens).
- A *word* (or *string*) over Σ is a finite sequence $w = w_1 w_2 w_3 ... w_n$, with each $w_i \in \Sigma$.
  - The *empty word* ϵ is a word over any alphabet
  - The set of all words over Σ is typically denoted Σ^*
  - E.g., $01001 \in \{0, 1\}^*$, *embiggen* $\in \{a, ..., z\}^*$
- A *language* over Σ is a set of words over Σ
  - Integer literals form a language over $\{0, ..., 9, -\}$
  - The keywords of OCaml form a (finite) language over ASCII
  - Syntactically-valid Java programs forms an (infinite) language over Unicode

Regular expressions (regex)

- *Regular expressions* are a formal language for describing formal languages.
  - E.g., $0|(1(0|1)^*)$ recognizes the language of all binary sequences without leading zeros

Regular expressions (regex)

- *Regular expressions* are a formal language for describing formal languages.
  - E.g., $0|(1(0|1)^*)$ recognizes the language of all binary sequences without leading zeros
- Syntax of regular expressions:
  - ϵ is a regular expression
  - $\emptyset$ is a regular expression
  - For any letter $a \in \Sigma$, a is a regular expression
  - For any regular expressions R_1 and R_2, R_1R_2 and $R_1 \mid R_2$ are regular expressions
  - For any regular expression R, R^* and (R) are regular expressions

Regular expressions (regex)

- *Regular expressions* are a formal language for describing formal languages.
  - E.g., $0|(1(0|1)^*)$ recognizes the language of all binary sequences without leading zeros
- Syntax of regular expressions:
  - ϵ is a regular expression
  - $\emptyset$ is a regular expression
  - For any letter $a \in \Sigma$, a is a regular expression
  - For any regular expressions R_1 and R_2, $R_1 R_2$ and $R_1 \mid R_2$ are regular expressions
  - For any regular expression R, R^* and (R) are regular expressions
- Meaning of regular expressions:

$$\begin{aligned}
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(a) &= \{a\} \\
\mathcal{L}(R_1 R_2) &= \{uv : u \in \mathcal{L}(R_1) \wedge v \in \mathcal{L}(R_2)\} \\
\mathcal{L}(R_1 | R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R^*) &= \{\epsilon\} \cup \mathcal{L}(R) \cup \mathcal{L}(RR) \cup \mathcal{L}(RRR) \cup \ldots
\end{aligned}$$

- Regular languages are useful in compilers – typically used in the *lexing* phase. Examples of regular languages:
  - Keywords
  - Integer / floating point constants
  - Identifiers
  - ...

- Regular languages are useful in compilers – typically used in the *lexing* phase. Examples of regular languages:
  - Keywords
  - Integer / floating point constants
  - Identifiers
  - ...
- But not always enough!. Examples of *non*-regular languages:
  - The language of regular expressions
  - Syntactically valid C expressions
  - Syntacically valid OCaml programs
  - ...

The language of regular expressions

- Recall syntax of regular expressions:
  - ϵ is a regular expression
  - $\emptyset$ is a regular expression
  - For any letter $a \in \Sigma$, a is a regular expression
  - For any regular expressions R_1 and R_2, $R_1 R_2$ and $R_1 \mid R_2$ are regular expressions
  - For any regular expression R, R^* and (R) are regular expressions
- *Context-free grammar* for regular expressions:

<regexp> ::= ϵ
| $\emptyset$
| a for each $a \in \Sigma$
| <regexp><regexp> | <regexp>|<regexp>
| <regexp>* | (<regexp>)

Anatomy of context-free grammars

- `<regexp>` is a *non-terminal* symbol.
- $\epsilon, \emptyset, a, |, *, (,)$ are *terminal* symbols.
- Grammar consists of a set of rules, written as

  ```
  <nonterminal> ::= sequence of terminal and nonterminal symbols
  ```

  - | abbreviates multiple productions w/ same left-hand side
    - `<regexp> ::= (<regexp>) | <regexp>*` means
      `<regexp> ::= (<regexp>)`
      `<regexp> ::= <regexp>*`
- Grammar recognizes the set of words *over the terminal symbols* that can be obtained from a designated *non-terminal* by (repeatedly) replacing the left-hand side of some rule with its right-hand side
  - `<regexp>` $\rightarrow$ `<regexp><regexp>` $\rightarrow$ `a<regexp>` $\rightarrow$ `a<regexp>*` $\rightarrow$ `ab*`

- Thursday's lecture: x86lite
  - Simple subset of x86 (~20 instructions)
  - Suitable as a compilation target for Oat
- HW1 on canvas. Due Feb 9.
  - You will implement:
    - A simulator for X86lite machine code
    - An assembler
    - A loader
  - You may work individually or in pairs