CS443: Compiler Construction

Lecture 0
What happens when you call gcc?

This class

- foo.c
- foo.s
- lib1.o
- lib2.o
- foo.o
- ...
There are different ways of translating a programming language

- **Source Code**
  - Compiler
  - Binary/Assembly
    - Ex.: C, C++

- **Source Code**
  - Interpreter
    - Ex.: Python

- **Source Code**
  - Compiler
  - Bytecode
    - VM
    - Ex.: Java
Compilers translate code in phases

**“Front End”**

- **Source Code**
  - `a = b + c - 1`
- **Lexical Analyzer**
- **Tokens**
  - `VAR a`
  - `EQUAL`
  - `VAR b`
  - `OP +`
  - `VAR C`
  - `OP -`
  - `CONST 1`
- **Parser**
- **Abstract Syntax**
- **Lowering**
- **Analysis**
  - Assign
    - `a +`
    - `b -`
    - `c 1`
  - `temp = c - 1`
  - `a = b + temp`

**“Back End”**

- **Intermed. Rep.**
- **Code Gen.**
- **Target Code**
  - `subl %rax, 1`
  - `addl %rax, %rbx`
May have many more phases, several intermediate representations
Front End is language specific  
Back End is machine specific

```
a = b + c - 1
VAR a
EQUAL
VAR b
OP +
VAR C
OP -
CONST 1
```

```
Assign
a +

b -
c 1
```

```
temp = c - 1
a = b + temp
```
Can (and usually do) swap out back ends to target different machines

Machine-Independent Optimizations

Intermediate Representation

x86
ARM
PowerPC

...
Compiler collections also swap out front ends for different languages.

Machine-Independent Optimizations

Intermediate Representation

C

C++

Java

... x86

ARM

PowerPC

...
A Small ML Compiler

- Higher-order
  - Typed
  - Structured Data
  - Nested Expressions
  - Unlimited Variables
- First-order
  - Typed
  - Structured Data
  - Nested Expressions
  - Unlimited Variables
- First-order
  - Typed
  - Structured Data
  - Flat Expressions
  - Unlimited Variables
- First-order
  - Untyped
  - No Structured Data
  - Flat Expressions
  - 32 Hardware Registers

Source Code → MiniCaml AST → Mini-C → LLVM → LLVM → Risc-V

Lexing/Parsing → Closure Conv./Lifting → IR Generation → Register Allocation → Instruction Selection

Optimization
Projects

• 7 projects, 2-3 weeks each (Except Project 0, Due 8/30)
• Mostly (entirely?) programming – graded with automated tests
• Work individually or in pairs
• Handed out + submitted on Blackboard

Late Days:
• 6 per student, extend deadline 24 hours
• No more than 2 per assignment
• If a pair, must both use a late day
Background

• Prerequisite: CS440 (Programming Languages and Translators)
  • Lexing and Parsing (Will review)
  • Abstract syntax, working with ASTs (will review very briefly today)
  • Type checking/inference
  • Statically-typed functional programming
    • Specifically, OCaml. If you know Haskell or Racket, can learn it quickly.
    • Haskell w/o monads
    • Racket w/ way fewer parens
• If you’re not familiar with the above, I suggest brushing up in the next couple weeks.
Websites to know

• Course website: http://cs.iit.edu/~smuller/cs443/
  • Full syllabus/policies/schedule/lecture notes. Go there.
• Blackboard
• Discord
Exams

• Midterm (Oct. 25)
• Final Exam (during finals week, schedule posted by Registrar)
• Open book, open notes
Grading

• 50% Projects
• 20% Midterm
• 30% Final
Textbooks

• Appel. *Modern Compiler Implementation in ML* (Highly recommended) (Also have C, Java versions)

If you already have it:

Academic Honesty

• Submitted solutions must be your own work (and your partner)
• Can discuss course concepts with other students, but don’t share/look at code.
• If using online resources/code:
  • Don’t search for code that substantially solves the assigned problem. Be reasonable.
  • If using small snippets of code, cite them (e.g., URL in a comment)
OK, back to programming languages

First-order
Typed
No Structured Data
Nested Expressions
Unlimited Variables
Simple
Easy to compile

IITRAN

IITRAN/7040 – 1964
IITRAN/360 - 1966
Abstract Syntax

• BNF (Backus-Naur Form)

```
type ::= INTEGER | CHARACTER | LOGICAL
bop ::= + | - | * | / | <-
uop ::= ~ | NOT | INT | CH | LG
exp ::= x | num | char | exp bop exp | uop exp
stmt ::= STOP | IF exp THEN stmt (ELSE stmt) | WHILE exp stmt
        | DO stmtlist | type varlist
varlist ::= x | x varlist
stmtlist ::= stmt | stmt stmtlist
```

Type casts

Not actually BNF, but you know what we mean
Abstract Syntax Trees (ASTs)

IF X + 2 < 5 DO … ELSE …

IF

<

DO

ELSE

...

...

+ 5

X 2
AST definition in OCaml

```ocaml
typedef var = string
typedef typ = TLogical | TCharacter | TInteger
typedef bop = BAdd | BSub | BMul | BDiv | BAnd | BOr | BGt (* > *)
         | BGe (* >= *) | BLt (* < *) | BLe (* <= *) | BNe (* # *) | BEq (* = *)
typedef unop = UNeg | Unot (* type conversions *) | UChar | ULog | UReal | Uint
typedef const = CChar of char | CInt of int | CLog of bool
typedef exp = EConst of const | EVar of var | EBinop of bop * exp * exp
              | EAssign of exp * exp | EUunop of unop * exp
typedef stmt = SDecl of typ * spec list | SDo of stmt list
             | SExp of exp
             | SIf of exp * stmt * (* ELSE *) stmt option
             | SWhile of exp * stmt | SStop
```
Pattern matching on an AST in OCaml

```ocaml
let print_const c =
match c with
| CChar c -> Printf.printf "\%c\n" c
| CInt n -> Printf.printf "\%d\n" n
| CLog true -> Printf.printf "true\n"
| CLog false -> Printf.printf "false\n"
```
Example: type checking

```plaintext
let rec typecheck_exp (ctx: ctx) (e: exp) : typ =
match e with
| EConst c -> typecheck_const c
| EVar v -> lookup ctx v
| EBinop (b, e1, e2) ->
  let (e1t, e2t) = (typecheck_exp ctx e1, typecheck_exp ctx e2) in
  let (etype, rtype) =
    match b with
    | BAdd | BSub | BMul | BDiv -> (TInteger, TInteger)
    | BGt | BGe | BLt | BLe | BNe | BEq -> (TInteger, TLogical)
    | BAnd | BOr -> (TLogical, TLogical)
  in
  (if e1t <> e2t then raise (TypeError (e1t, e2t))
   else if e1t = etype then ()
   else raise (TypeError (etype, e1t)))
  rtype
```

...
Abstract Syntax is not Concrete Syntax

```
IF X + 2 < 5 DO ... ELSE ...
```

```
+  DO  ELSE

  X  <  ...

  2  5  ...
```