CS 516: COMPILERS

Lecture 2

Topics

- How to represent programs as data structures.
- How to write programs that process programs.

Materials

- lec02.zip (simple.ml, translate.ml, etc.)

INTERPRETERS

Factorial: Everyone's Favorite Function

- Consider this implementation of factorial in a hypothetical programming language that we'll call SIMPLE:
  (**S**imple **I**mperative **P**rogramming **L**anguag**E**)

```
X = 6;
ANS = 1;
whileNZ (x) {
    ANS = ANS * X;
    X = X + -1;
}
```

- We need to describe the constructs of this hypothetical language
  - ***Syntax***: which sequences of characters count as a legal "program"?
  - ***Semantics***: what is the meaning (behavior) of a legal "program"?

"Object" vs. "Meta" language

Object language: the language (syntax / semantics) being described or manipulated	**Metalanguage:** the language (syntax / semantics) used to **describe** some object language
Today: SIMPLE	Interpreter written in OCaml
Course project: OAT → LLVM → x86_64	Compiler written in OCaml
Clang compiler: C/C++ → LLVM → x86_64	Compiler written in C++
Metacircular interpreter: lisp	Interpreter written in lisp

Grammar for a Simple Language

```
<exp>  ::=
       |     <X>
       |     <exp> + <exp>
       |     <exp> * <exp>
       |     <exp> < <exp>
       |     <integer constant>
       |     (<exp>)

<cmd>  ::=
       |     skip
       |     <X> = <exp>
       |     ifNZ <exp> { <cmd> } else { <cmd> }
       |     whileNZ <exp> { <cmd> }
       |     <cmd>; <cmd>
```

- Concrete syntax (grammar) for a simple imperative language
  - Written in "Backus-Naur form"
  - <*exp*> and <*cmd*> are *nonterminals*
  - '::=' , '|' , and <…> symbols are part of the *meta* language
  - keywords, like '`skip`' and '`ifNZ`' and symbols, like '`{`' and '+' are part of the *object* language

- Need to represent the *abstract syntax* (i.e. hide the irrelevant of the concrete syntax)
- Implement the *operational semantics* (i.e. define the behavior, or meaning, of the program)

Demo Interpreter/Compiler

1. Interpreter – `simple.ml (simple-soln.ml)`
2. Compiler – `translate.ml`

```
cd simple
dune build
dune exec bin/simple.exe

dune utop
utop # #use “bin/simple.ml";;
utop # let s' = (interpret_cmd init_state factorial);;
utop # lookup s’ “ANS”;;
```