Lecture 13

COMPILER DESIGN

Announcements

- **HW 3:** due soon

- **HW 4:** Building a frontend
  - Work with lexer and parser generators
  - Compile a C-like source language to LLVM

Debugging parser conflicts.
Disambiguating grammars.

MENHIR IN PRACTICE

Practical Issues

- Dealing with source file location information
  - In the lexer and parser
  - In the abstract syntax

  - See range.ml, ast.ml

- Lexing comments/strings
  - See lexer.mll

Menhir output

- You can get verbose ocamlyacc debugging information by doing
  - `menhir --explain …`
  - or, if using ocamlbuild
    `ocamlbuild –use-menhir -yaccflag ––explain …`

- The result is a <basename>.conflicts file describing the error
  - The parser items of each state use the '.' just as described above

- The flag --dump generates a full description of the automaton

- Example: start-parser.mly

Precedence and Associativity Declarations

- Parser generators often support precedence/associativity declarations
  - Hints to the parser about how to resolve conflicts
  - See good-parser.mly
- Pros
  - Avoids having to manually resolve those ambiguities by manually introducing extra nonterminals (as seen in hand-parser.mly)
  - Easier to maintain the grammar
- Cons
  - Can't as easily re-use the same terminal (if associativity differs)
  - Introduces another level of debugging
- Limits
  - Not always easy to disambiguate just with precedence/associativity

Example Ambiguity in Real Languages

- Consider this grammar
  $S \mapsto$ `if (E)` S
  $S \mapsto$ `if (E)` S `else` S
  $S \mapsto X = E$
  $E \mapsto \ldots$

- Is this grammar OK?

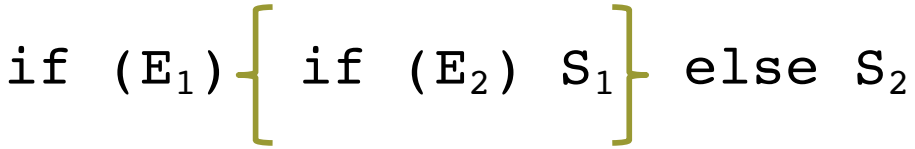
- Consider how to parse

  `if (`E_1`)` `if (`E_2`)` S_1 `else` S_2

- The "dangling else" problem
- Which is the "right" answer?
- How to change the grammar?

How to disambiguate if-then-else

- Want to rule out



- Observation: An un-matched '`if`' should not appear as the '`then`' clause of a containing '`if`'

S ↦ M | U // M = "matched", U = "unmatched"
U ↦ `if (`E`)` S // Unmatched 'if'
U ↦ `if (`E`)` M `else` U // Nested if is matched
M ↦ `if (`E`)` M `else` M // Matched 'if'
M ↦ X = E // Other statements

- See else-resolved-parser.mly

Alternative: Use { }

- Ambiguity arises because the 'then' branch is not well bracketed

```
if (E1) { if (E2) { S1 } } else S2      // unambiguous
if (E1) { if (E2) { S1 } else S2 }      // unambiguous
```

- So, one could just require brackets
  - But requiring them for the else clause too leads to ugly code for chained if-statements

```
if (c1) {
  …
} else {
  if (c2) {

  } else {
    if (c3) {

    } else {

    }
  }
}
```

So, compromise? Allow unbracketed else block only if the body is 'if'

```
if (c1) {

} else if (c2) {

} else if (c3) {

} else {

}
```

Benefits
- Less ambiguous
- Easy to parse
- Enforces good style

See HW4

OAT V 1.0

OAT

- Simple C-like imperative language
  - Supports 64-bit integers, arrays, strings
  - Top-level, mutually recursive procedures
  - Scoped local, imperative variables

- See examples in HW#4 later

- How to design/specify such a language?
  - Grammatical constructs
  - Semantic constructs

Compilation in a Nutshell

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

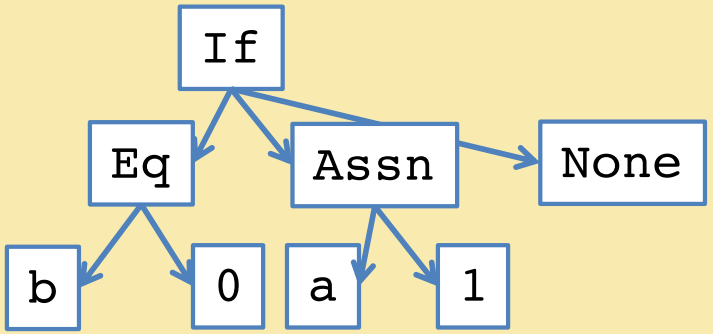
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Parsing

Abstract Syntax Tree:



Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b,
0
  br i1 %cnd, label %l2,
label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Analysis & Transformation

Backend

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

Untyped lambda calculus
Substitution
Evaluation

FIRST-CLASS FUNCTIONS

"Functional" languages

- Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
  - Functions can be passed as arguments (e.g. map or fold)
  - Functions can be returned as values (e.g. compose)
  - Function nest:

    Inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?
  - In an interpreter? In a compiled language?

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e
- It has variables, functions, and function application
  - That's it!
  - It's Turing Complete
  - It's the foundation for a *lot* of research in programming languages
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml

```
type exp =
  | Var of var          (* variables                 *)
  | Fun of var * exp    (* functions: fun x -> e *)
  | App of exp * exp    (* function application   *)
```

Concrete syntax

exp ::=	
| x	*variables*
| **fun** x -> exp	*functions*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

Values and Substitution

- The only values of the lambda calculus are (closed) functions

```
val ::=
      | fun x -> exp      functions are values
```

- To *substitute* value `v` for variable `x` in expression `e`
  - Replace all *free occurrences* of `x` in `e` by `v`
  - In OCaml: written `subst v x e`
  - In Math: written e{v/x}

- Function application is interpreted by *substitution*

```
  (fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

Lambda Calculus Operational Semantics

- Substitution function (in Math)

$x\{v/x\}$	$= v$	*(replace the free x by v)*
$y\{v/x\}$	$= y$	*(assuming $y \neq x$)*
$(\text{fun } x \text{ -> } exp)\{v/x\}$	$= (\text{fun } x \text{ -> } exp)$	*(x is bound in exp)*
$(\text{fun } y \text{ -> } exp)\{v/x\}$	$= (\text{fun } y \text{ -> } exp\{v/x\})$	*(assuming $y \neq x$)*
$(e_1\ e_2)\{v/x\}$	$= (e_1\{v/x\}\ e_2\{v/x\})$	*(substitute everywhere)*

- Examples

x y {(fun z ->z)/y} ⇒ x (fun z -> z)

(fun x -> x y){(fun z -> z) / y} ⇒ (fun x -> x (fun z -> z))

(fun x -> x){(fun z -> z) / x} ⇒ (fun x -> x) // x is not free!

Free Variables and Scoping

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by `add`

- We say variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say variable `y` is *bound* by "`fun y`"
  - Its scope is the body "`x + y`" in `fun y -> x + y`

- A term with no free variables is called *closed*
- A term with one or more free variables is called *open*

Free Variable Calculation

- OCaml code to compute the set of free variables in lambda expressions

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x         -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- Lambda expression `e` is *closed* if `free_vars e` is `VarSet.empty`
- In math notation

$$
\begin{aligned}
fv(x) &= \{x\} \\
fv(\text{fun } x \text{ -> } exp) &= fv(exp) \setminus \{x\} \quad \textit{(`x' is a bound in exp)} \\
fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2)
\end{aligned}
$$

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables

  (fun x -> (x y)) {(fun z -> x) / y} Note: x is free in (fun z -> x)
  = fun x -> (x (fun z -> x)) free x is *captured!!*

- Usually *not* the desired behavior
  - This property is sometimes called "dynamic scoping"

    The meaning of "x" is determined by where it is bound dynamically (not where it is bound statically)

  - Some languages (e.g. Emacs Lisp) are implemented with this as a "feature"
  - But, leads to hard to debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter

  (fun x -> y x) the "same" as (fun z -> y z)

- Two terms that differ only by consistent renaming of bound variables are called *alpha equivalent*

- The names of free variables do matter

  (fun x -> y x) *not* the "same" as (fun x -> z x)

  Intuitively: y an z may refer to different things from some outer scope

Fixing Substitution

- Consider the substitution operation

$$e_1\{e_2/x\}$$

  - To avoid capture, define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2
  - Then do the "naïve" substitution

- Example

(fun x -> (x y)) {(fun z -> x) / y}

= (fun x' -> (x' (fun z -> x)) *rename* x to x'

Operational Semantics

- Specified with 2 inference rules with judgments of the form exp ⇓ v
  - Read this notation as "program exp evaluates to value v"
  - We give a *call-by-value* semantics

    Function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

See fun.ml

IMPLEMENTING THE INTERPRETER