Course Objectives

- New programming paradigm
  - Functional programming
  - Environments and Closures
  - Patterns of Recursion
  - Continuation Passing Style
- Phases of an interpreter / compiler
  - Lexing and parsing
  - Type systems
  - Interpretation
- Programming Language Semantics
  - Lambda Calculus
  - Operational Semantics
  - Axiomatic Semantics

Major Phases of a Compiler

1. Source Program
2. Lex
3. Token
4. Parse
5. Abstract Syntax
6. Semantic Analysis
7. Environment
8. Translate
9. Intermediate Representation (CPS)
10. Assembler
11. Analyze + Transform
12. Optimized IR (CPS)
13. Instruction Selection
14. Instruction Optimize
15. Optimized Machine-Specific Assembly Language
16. Emit code
17. Unoptimized Machine-Specific Assembly Language
18. Relocatable Object Code
20. Interpreter Execution
21. Program Run
22. Source Program
23. Tokens
24. Parse
25. Abstract Syntax
26. Semantic Analysis
27. Environment
28. Translate
29. Intermediate Representation (CPS)

Meta-discourse

Language Syntax and Semantics
- Syntax
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- Semantics
  - Natural Semantics
  - Transition Semantics

Where We Are Going Next?

- We want to turn strings (code) into computer instructions
- Done in phases
- Break the big strings into tokens (lex)
- Turn tokens into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile)
Syntax of English Language

- **Pattern 1**
  - Subject | Verb
  - David   | sings
  - The dog | barked
  - Susan   | yawned

- **Pattern 2**
  - Subject | Verb | Direct Object
  - David   | sings | ballads
  - The professor | wants | to retire
  - The jury | found | the defendant guilty

Elements of Syntax

- **Character set** – previously always ASCII, now often 64 character sets
- **Keywords** – usually reserved
- **Special constants** – cannot be assigned to
- **Identifiers** – can be assigned to
- **Operator symbols**
- **Delimiters** (parenthesis, braces, brackets)
- **Blanks** (aka white space)

Elements of Syntax

- **Expressions**
  - if ... then begin ... ; ... end else begin ... ; ... end
- **Type expressions**
  - typexpr₁ -> typexpr₂
- **Declarations** (in functional languages)
  - let pattern = expr
- **Statements** (in imperative languages)
  - a = b + c
- **Subprograms**
  - let pattern₁ = expr₁ in expr

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing**: Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - **Parsing**: Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- **Regular expressions, regular grammars, finite state automata**
- **Context-free grammars, BNF grammars, syntax diagrams**
- **Whole family more of grammars and automata – covered in automata theory**
Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

Regular Expressions - Review

- Start with a given character set – a, b, c…
- Each character is a **regular expression**
  - It represents the set of one string containing just that character

Regular Expressions

- If x and y are regular expressions, then xy is a regular expression
  - It represents the set of all strings made from first a string described by x then a string described by y
    - If x(=a,ab) and y(=c,d) then xy(=ac,ad,abc,abd).
- If x and y are regular expressions, then x ∨ y is a regular expression
  - It represents the set of strings described by either x or y
    - If x(=a,ab) and y(=c,d) then x ∨ y(=a,ab,c,d)

Example Regular Expressions

- (0|1)*1
  - The set of all strings of 0’s and 1’s ending in 1, {1, 01, 11,…}
- a*b(a*)
  - The set of all strings of a’s and b’s with exactly one b
- ((01) v(10))*
  - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z) (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z ∨ 0 ∨ 1 ∨ … ∨ 9)*
  - Digit = (0 ∨ 1 ∨ … ∨ 9)
  - Number = 0 ∨ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)* ∨
    - (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)*
  - Keywords: if = if, while = while,…
Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata
- Should have seen in CS374

Lexing

- Different syntactic categories of “words”: tokens

Example:

- Convert sequence of characters into sequence of strings, integers, and floating point numbers.
- "asd 123 jkl 3.14" will become:
  [String "asd"; Int 123; String "jkl"; Float 3.14]

Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml

How to do it

To use regular expressions to parse our input we need:

- Some way to identify the input string — call it a lexing buffer
- Set of regular expressions,
- Corresponding set of actions to take when they are matched.

How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.

Mechanics

- Put table of reg exp and corresponding actions (written in ocaml) into a file <filename>.mll
- Call
  ocamllex <filename>.mll
- Produces Ocaml code for a lexical analyzer in file <filename>.ml
Sample Input

```ocaml
rule main = parse
  ['0'-'9']+ { print_string "Int\n"}
| ['0'-'9']+'.'['0'-'9']+ { print_string "Float\n"}
| ['a'-'z']+ { print_string "String\n"}
| _ { main lexbuf }
{
  let newlexbuf = (Lexing.from_channel stdin) in
  print_string "Ready to lex.\n";
  main newlexbuf
}
```

General Input

```ocaml
{ header }
let ident = regexp ...
rule entrypoint [arg1... argn] = parse
  regexp { action }
  | ...
  | regexp { action }
and entrypoint [arg1... argn] = parse ...
and ...
{ trailer }
```

Ocamllex Input

- `header` and `trailer` contain arbitrary ocaml code put at top and bottom of `<filename>.ml`
- `let ident = regexp ...` Introduces `ident` for use in later regular expressions

Ocamllex Regular Expression

- Single quoted characters for letters: ‘a’
- ‘_’ (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “string”: concatenation of sequence of characters
- `e1 / e2`: choice - what was `e1 ∨ e2`
- `[c1 - c2]`: choice of any character between first and second inclusive, as determined by character codes
- `[^c1 - c2]`: choice of any character NOT in set
- `e*`: same as before
- `e+`: same as `e e*`
- `e?`: option - was `e1 ∨ ε`
Ocamllex Regular Expression

- $e_1 \# e_2$: the characters in $e_1$ but not in $e_2$; $e_1$ and $e_2$ must describe just sets of characters
- **ident**: abbreviation for earlier reg exp in let
- **e as id**: binds the result of $e$ to $id$ to be used in the associated action

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html

Example: test.mll

```ml
{   type result = Int of int | Float of float | String of string
}
let digit = ['0'-'9']
let digits = digit+
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter+
```

Example: test.mll

```ml
rule main = parse
  (digits)'.'digits as f
  { Float (float_of_string f) }
| digits as n   { Int (int_of_string n) }
| letters as s  { String s}
| _             { main lexbuf }
{
  let newlexbuf = (Lexing.from_channel stdin) in
  print_string "Ready to lex."
  print_newline ();
  main newlexbuf
}
```

Example

```ml
# #use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf ->
  int -> result = <fun>
Ready to lex.
hi there 234 5.2
- : result = String "hi"

What happened to the rest?!!?
```

Example

```ml
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```
Problem

- How to get lexer to look at more than the first token at one time?
- Answer: action has to tell it to -- recursive calls
- Side Benefit: can add “state” into lexing
- Note: already used this with the _ case

Example

Example Results

Ready to lex.
hi there 234 5.2
- : result list = [String "hi"; String "there"; Int 234; Float 5.2]
#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

(* Continued from rule main *)
| open_comment { comment lexbuf}
| eof { [] }  
| _ { main lexbuf }

and comment = parse
  close_comment { main lexbuf }
  | _ { comment lexbuf }

Dealing with nested comments

rule main = parse ...
| open_comment { comment 1 lexbuf}
| eof { [] }
| _ { main lexbuf }

and comment depth = parse
  open_comment { comment (depth+1) lexbuf }
  close_comment if depth = 1 then main lexbuf
  else comment (depth - 1) lexbuf
  | _ { comment depth lexbuf }
Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata
- Pushdown automata

Whole family more of grammars and automata – covered in automata theory

BNF Grammars

- Start with a set of characters, \( a, b, c, \ldots \)
  - We call these *terminals*
- Add a set of different characters, \( X, Y, Z, \ldots \)
  - We call these *nonterminals*
- One special nonterminal \( S \) called *start symbol*

BNF Grammars

- BNF rules (aka *productions*) have form
  \[ X ::= y \]
  where \( X \) is any nonterminal and \( y \) is a string of terminals and nonterminals
- BNF grammar is a set of BNF rules such that every nonterminal appears on the left of some rule

Example: Regular Grammars

- Regular grammar:
  \[
  \langle \text{Balanced} \rangle ::= \varepsilon \\
  \langle \text{Balanced} \rangle ::= 0\langle \text{OneAndMore} \rangle \\
  \langle \text{Balanced} \rangle ::= 1\langle \text{ZeroAndMore} \rangle \\
  \langle \text{OneAndMore} \rangle ::= 1\langle \text{Balanced} \rangle \\
  \langle \text{ZeroAndMore} \rangle ::= 0\langle \text{Balanced} \rangle
  \]
- Generates even length strings where every initial substring of even length has same number of 0’s as 1’s

Example of BNF: Regular Grammars

- Subclass of BNF -- has only rules of the form:
  \[
  \langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle \langle \text{nonterminal} \rangle \text{ or } \\
  \langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle \text{ or } \\
  \langle \text{nonterminal} \rangle ::= \varepsilon
  \]
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
- Close connection to nondeterministic finite state automata
  - nonterminals = states;
  - rule = edge

BNF Grammars

- BNF rules (aka *productions*) have form
  \[ X ::= y \]
  where \( X \) is any nonterminal and \( y \) is a string of terminals and nonterminals
- BNF grammar is a set of BNF rules such that every nonterminal appears on the left of some rule
Sample BNF Grammar

- Language: Parenthesized sums of 0’s and 1’s
- `<Sum>` ::= 0
- `<Sum>` ::= 1
- `<Sum>` ::= `<Sum>` + `<Sum`
- `<Sum>` ::= (<Sum>)

Sample Grammar

- Terminals: 0 1 + ( )
- Nonterminals: `<Sum>`
- Start symbol = `<Sum>`
- `<Sum>` ::= 0
- `<Sum>` ::= 1
- `<Sum>` ::= `<Sum>` + `<Sum`
- `<Sum>` ::= (<Sum>)
- Can be abbreviated as
  `<Sum>` ::= 0 | 1
  | `<Sum>` + `<Sum>` | (<Sum>)

BNF Derivations

- Given rules
  \[ X ::= \text{yZw} \text{ and Z ::= } \nu \]
we may replace Z by \( \nu \) to say
  \[ X \Rightarrow \text{yZw} \Rightarrow \gamma \nu \nu \]
- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal

BNF Derivations

- Pick a non-terminal
  `<Sum>` =>
BNF Derivations

- Pick a non-terminal:
  \[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]

- Pick a non-terminal:
  \[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]

- Pick a non-terminal:
  \[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]

- Pick a non-terminal:
  \[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]

BNF Derivations

- Pick a rule and substitute:
  \[ <\text{Sum}> ::= ( <\text{Sum}> ) \]
  \[ <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]

- Pick a rule and substitute:
  \[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]
  \[ <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]

- Pick a rule and substitute:
  \[ <\text{Sum}> ::= 1 \]
  \[ <\text{Sum}> + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
  \[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
BNF Derivations

- Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]

BNF Derivations

- Pick a rule and substitute:

- \( <\text{Sum}> ::= 0 \)

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + 0 \]

BNF Derivations

- Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + 0 \]

BNF Derivations

- Pick a rule and substitute:

- \( <\text{Sum}> ::= 0 \)

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + 0 \]

BNF Derivations

- \( (0 + 1) + 0 \) is generated by grammar

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + 0 \]
\[ \Rightarrow (0 + 1) + 0 \]

Parse Trees

- Graphical representation of derivation

- Each node labeled with either non-terminal or terminal

- If node is labeled with a terminal, then it is a leaf (no sub-trees)

- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it
Example

- Consider grammar:
  \[
  \text{\texttt{exp}} ::= \text{\texttt{factor}} \\
  \text{\texttt{factor}} ::= \text{\texttt{bin}} \\
  \text{\texttt{bin}} ::= 0 | 1
  \]

- Goal: Build parse tree for \(1 \times 1 + 0\) as an \texttt{exp}

Example cont.

- \(1 \times 1 + 0\):
  \[
  \text{\texttt{exp}}
  \]
  \[
  \text{\texttt{factor}}
  \]

  Use rule: \texttt{exp} ::= \texttt{factor}

Example cont.

- \(1 \times 1 + 0\):
  \[
  \text{\texttt{exp}}
  \]
  \[
  \text{\texttt{factor}}
  \]
  \[
  \text{\texttt{bin}} * \text{\texttt{exp}}
  \]

  Use rules: \texttt{bin} ::= 1 and \texttt{exp} ::= \texttt{factor} + \texttt{factor}

Example cont.

- \(1 \times 1 + 0\):
  \[
  \text{\texttt{exp}}
  \]
  \[
  \text{\texttt{factor}}
  \]
  \[
  \text{\texttt{bin}}
  \]

  Use rule: \texttt{factor} ::= \texttt{bin}
Example cont.

$1 \cdot 1 + 0$:

```
<exp>
 | <factor>
 |   <bin> 1
 |     <exp>
 |       <factor> + <factor>
 |         <bin> 1 <bin> 0
```

Use rules: $<bin> ::= 1 \mid 0$

Example cont.

$1 \cdot 1 + 0$:

```
<exp>
 | <factor>
 |   <bin> 1
 |     <exp>
 |       <factor> + <factor>
 |         <bin> 1 <bin> 0
```

Use rules: $<bin> ::= 1 \mid 0$