Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Three Main Topics of the Course

I  New Programming Paradigm

II  Language Translation

III  Language Semantics
II : Language Translation

Type Systems

Lexing and Parsing

Interpretation
Major Phases of a Compiler

Source Program

- Lex
- Parse
- Abstract Syntax
- Semantic Analysis
- Translate
- Intermediate Representation

Optimize

- Optimized IR
- Instruction Selection
- Unoptimized Machine-Specific Assembly Language
- Optimize
- Optimized Machine-Specific Assembly Language
- Emit code
- Assembly Language
- Assembler

- Relocatable Object Code
- Linker
- Machine Code

Modified from “Modern Compiler Implementation in ML”, by Andrew Appel
Where We Are Going Next?

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile)
Meta-discourse

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

- Syntax is the description of which strings of symbols are meaningful expressions in a language.
- It takes more than syntax to understand a language; need meaning (semantics) too.
- Syntax is the entry point.
Syntax of English Language

- **Pattern 1**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
</tr>
<tr>
<td>The dog</td>
<td>barked</td>
</tr>
<tr>
<td>Susan</td>
<td>yawned</td>
</tr>
</tbody>
</table>

- **Pattern 2**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
<th>Direct Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
<td>ballads</td>
</tr>
<tr>
<td>The professor</td>
<td>wants</td>
<td>to retire</td>
</tr>
<tr>
<td>The jury</td>
<td>found</td>
<td>the defendant guilty</td>
</tr>
</tbody>
</table>
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
  
  \[ \text{if ... then begin ... ; ... end else begin ... ; ... end} \]

- Type expressions
  
  \[\text{typexpr}_1 \rightarrow \text{typexpr}_2\]

- Declarations (in functional languages)
  
  \[\text{let pattern} = \text{expr}\]

- Statements (in imperative languages)
  
  \[a = b + c\]

- Subprograms
  
  \[\text{let pattern}_1 = \text{expr}_1 \text{ in expr}\]
Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)
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
  - \( L(a) = \{a\} \)
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 $L(x)=\{a,ab\}$ and $L(y)=\{c,d\}$
then $L(xy) =\{ac,ad,abc,abd\}$
If \( x \) and \( y \) are regular expressions, then \( x \lor y \) is a regular expression

It represents the set of strings described by either \( x \) or \( y \)

If \( L(x) = \{a, ab\} \) and \( L(y) = \{c, d\} \)
then \( L(x \lor y) = \{a, ab, c, d\} \)
Regular Expressions

- If $x$ is a regular expression, then so is $(x)$
  - It represents the same thing as $x$
- If $x$ is a regular expression, then so is $x^*$
  - It represents strings made from concatenating zero or more strings from $x$
    - If $L(x) = \{a, ab\}$ then $L(x^*) = \{\text{""}, a, ab, aa, aab, abab, \ldots\}$
- $\varepsilon$
  - It represents $\{\text{""}\}$, set containing the empty string
- $\emptyset$
  - It represents $\{\ \}$, the empty set
Example Regular Expressions

- \((0 \lor 1)^* 1\)
  - The set of all strings of 0’s and 1’s ending in 1, \{1, 01, 11, \ldots\}

- \(a^*b(a^*)\)
  - The set of all strings of a’s and b’s with exactly one b

- \(((01) \lor (10))^*\)
  - You tell me

- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words
Regular Grammars

- Subclass of BNF (covered in detail soon)
- Only rules of form
  \[ <\text{nonterminal}> ::= <\text{terminal}> <\text{nonterminal}> \text{ or } \]
  \[ <\text{nonterminal}> ::= <\text{terminal}> \text{ or } \]
  \[ <\text{nonterminal}> ::= \varepsilon \]
- 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 \(\cong\) states; rule \(\cong\) edge
Example

- Regular grammar:
  \[
  \texttt{<Balanced>} ::= \varepsilon \\
  \texttt{<Balanced>} ::= 0\texttt{<OneAndMore>} \\
  \texttt{<Balanced>} ::= 1\texttt{<ZeroAndMore>} \\
  \texttt{<OneAndMore>} ::= 1\texttt{<Balanced>} \\
  \texttt{<ZeroAndMore>} ::= 0\texttt{<Balanced>}
  \]

- Generates even length strings where every initial substring of even length has same number of 0’s as 1’s
Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = \((a \lor b \lor \ldots \lor z \lor A \lor B \lor \ldots \lor Z) (a \lor b \lor \ldots \lor z \lor A \lor B \lor \ldots \lor Z \lor 0 \lor 1 \lor \ldots \lor 9)^*\)
  - Digit = \((0 \lor 1 \lor \ldots \lor 9)\)
  - Number = \(0 \lor (1 \lor \ldots \lor 9)(0 \lor \ldots \lor 9)^* \lor \sim (1 \lor \ldots \lor 9)(0 \lor \ldots \lor 9)^*\)
  - 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

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

```plaintext
{ 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
Ocamlllex Input

- `<filename>.ml` contains one lexing function per `entrypoint`
  - Name of function is name given for `entrypoint`
  - Each entry point becomes an Ocaml function that takes $n+1$ arguments, the extra implicit last argument being of type `Lexing.lexbuf`
  - `arg1... argn` are for use in `action`
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
- \( e_1 / e_2 \): choice - what was \( e_1 \lor e_2 \)
- \([c_1 - c_2]\): choice of any character between first and second inclusive, as determined by character codes
- \([^c_1 - c_2]\): choice of any character NOT in set
- \(e^*\): same as before
- \(e^+\): same as \(e^*\)
- \(e?\): option - was \(e_1 \lor \varepsilon\)
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
  ```ocaml
  let ident = regexp
  ```
- $e_1$ as **id**: binds the result of $e_1$ to **id** to be used in the associated **action**
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

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

# 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?!?
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
Your Turn

- Work on MP4
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers
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

rule main = parse
   (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf }
| digits as n { Int (int_of_string n) :: main lexbuf }
| letters as s { String s :: main lexbuf }
| eof { [] }
| _ { main lexbuf }
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

First Attempt

let open_comment = "(*"
let close_comment = "*)"

rule main = parse
    (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
|  digits as n      { Int (int_of_string n) :: main lexbuf }
|  letters as s    { String s :: main lexbuf}
Dealing with comments

| 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 }
Dealing with nested comments

rule main = parse
    (digits) '.' digits as f { Float (float_of_string f) ::
    main lexbuf}
| digits as n          { Int (int_of_string n) :: main lexbuf }
| letters as s         { String s :: main lexbuf }
| open_comment         { (comment 1 lexbuf}
| eof                  { [] } 
| _ { main lexbuf }
Dealing with nested comments

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

- Whole family more of grammars and automata – covered in automata theory
Sample Grammar

- Language: Parenthesized sums of 0’s and 1’s

- \(<\text{Sum}> ::= 0\)
- \(<\text{Sum}> ::= 1\)
- \(<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>\)
- \(<\text{Sum}> ::= (<\text{Sum}>))\)
BNF Grammars

- Start with a set of characters, \(a,b,c,...\)
  - 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
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
  \[ X ::= yZw \text{ and } Z ::= v \]
  we may replace \( Z \) by \( v \) to say
  \[ X \rightarrow yZw \rightarrow yvww \]

- Sequence of such replacements called \textit{derivation}

- Derivation called \textit{right-most} if always replace the right-most non-terminal
BNF Derivations

- Start with the start symbol:

\(<\text{Sum}>\) =>
BNF Derivations

- Pick a non-terminal

\[ <\text{Sum}> \Rightarrow \]
Pick a rule and substitute:

- `<Sum> ::= <Sum> + <Sum>`

`<Sum> => <Sum> + <Sum>`
BNF Derivations

Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
BNF Derivations

- Pick a rule and substitute:
  - \( \langle \text{Sum} \rangle ::= ( \langle \text{Sum} \rangle ) \)
  - \( \langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \)
  - \( \Rightarrow ( \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \)
BNF Derivations

- Pick a non-terminal:

\[
<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>
\]

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

- Pick a rule and substitute:
  - `<Sum> ::= <Sum> + <Sum>`
  - `<Sum> => <Sum> + <Sum>`
  - `=> ( <Sum> ) + <Sum>`
  - `=> ( <Sum> + <Sum> ) + <Sum>`
BNF Derivations

Pick a non-terminal:

\[ \langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \]
\[ \Rightarrow ( \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \]
\[ \Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \]
BNF Derivations

Pick a rule and substitute:

- `<Sum> ::= 1`

```
<Sum>  =>  <Sum> + <Sum>

=> ( <Sum> ) + <Sum>

=> ( <Sum> + <Sum> ) + <Sum>

=> ( <Sum> + 1 ) + <Sum>
```
Pick a non-terminal:

\[ \langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \]
\[ \Rightarrow ( \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \]
\[ \Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \]
\[ \Rightarrow ( \langle \text{Sum} \rangle + 1 ) + \langle \text{Sum} \rangle \]
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 0`

  `<Sum> => <Sum> + <Sum>`

  `=> ( <Sum> ) + <Sum>`

  `=> ( <Sum> + <Sum> ) + <Sum>`

  `=> ( <Sum> + 1 ) + <Sum>`

  `=> ( <Sum> + 1 ) + 0`
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

\Rightarrow ( 0 + 1 ) + 0
BNF Derivations

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

\[ <Sum> \Rightarrow <Sum> + <Sum> \]
\[ \Rightarrow ( <Sum> ) + <Sum> \]
\[ \Rightarrow ( <Sum> + <Sum> ) + <Sum> \]
\[ \Rightarrow ( <Sum> + 1 ) + <Sum> \]
\[ \Rightarrow ( <Sum> + 1 ) + 0 \]
\[ \Rightarrow ( 0 + 1 ) + 0 \]
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

<Sum> =>