Programming Languages and Compilers (CS 421)

Sasa Misailovic
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha
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

Source Program

Lex

Tokens

Parse

Abstract Syntax

Semantic Analysis

Environment

Translate

Intermediate Representation (CPS)

Analyze + Transform

Optimized IR (CPS)

Instruction Selection

Unoptimized Machine-Specific Assembly Language

Instruction 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
Major Phases of a PicoML Interpreter

Source Program
   Lex
   Tokens
Parse
   Abstract Syntax
Semantic Analysis
   Environment
Translate
   Intermediate Representation (CPS)

Analyzer + Transform
Optimized IR (CPS)

Interpreter Execution
Program Run
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

<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**
  
  \[ \text{typexpr}_1 \to \text{typexpr}_2 \]

- **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
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 $x \lor y$ 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 $x \lor y=\{a,ab,c,d\}$
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 $x = \{a, ab\}$ then $x^* = \{\epsilon, a, ab, aa, aab, abab, \ldots\}$
- $\epsilon$
  - It represents $\{\epsilon\}$, 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
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

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

```latex
\{ header \}

let ident = regexp ...

rule entrypoint [arg1... argn] = parse
    regexp \{ action \}
    | ...
    | ... 
    | regexp \{ action \}

and entrypoint [arg1... argn] = parse ...
    and ...

\{ trailer \}
```
Ocamlleex 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 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 $e_1 \lor e_2$
Ocamllex Regular Expression

- \([c_1 - c_2]\): 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 \(e \ e^*\)
- \(e^?\): 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
- \textit{ident}: abbreviation for earlier reg exp in let \textit{ident} = \textit{regexp}
- \( e_1 \text{ as } id \): binds the result of \( e_1 \) to \textit{id} to be used in the associated \textit{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?!??
Example

```ocaml
# 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

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

(* 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 \textit{terminals}
- Add a set of different characters, \( X, Y, Z, \ldots \)
  - We call these \textit{nonterminals}
- One special nonterminal \( S \) called \textit{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:
  \[
  \text{<Balanced>} ::= \varepsilon \\
  \text{<Balanced>} ::= 0\text{<OneAndMore>} \\
  \text{<Balanced>} ::= 1\text{<ZeroAndMore>} \\
  \text{<OneAndMore>} ::= 1\text{<Balanced>} \\
  \text{<ZeroAndMore>} ::= 0\text{<Balanced>}
  \]

- 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:
  
  `<nonterminal>::=<terminal><nonterminal>` or
  `<nonterminal>::=<terminal>` or
  `<nonterminal>::=ε`

- 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
  \[ X ::= yZw \text{ and } Z ::= v \]
  we may replace \( Z \) by \( v \) to say
  \[ X \Rightarrow yZw \Rightarrow yvw \]
- 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:

<Sum> =>
BNF Derivations

- Pick a non-terminal

<Sum> =>
BNF Derivations

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

\(<\text{Sum}\> => \<\text{Sum}\> + \<\text{Sum}\>\)

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

- Pick a non-terminal:

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

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

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

- Pick a non-terminal:

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

- Pick a rule and substitute:
  - \(<\text{Sum} >:\ ::= \ 1\>

\(<\text{Sum}>\) \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 non-terminal:

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

=> ( <Sum> ) + <Sum>

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

=> ( <Sum> + 1 ) + <Sum>
Pick a rule and substitute:

- `<Sum>` ::= 0

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

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

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

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

=> ( `<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}>

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

$\Rightarrow ( 0 + 1 ) + 0$
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:
  
  \[
  \begin{align*}
  &<\text{exp}> ::= <\text{factor}> \\
  &\quad \quad | \quad <\text{factor}> + <\text{factor}> \\
  
  &<\text{factor}> ::= <\text{bin}> \\
  &\quad \quad | \quad <\text{bin}> \ast <\text{exp}> \\
  
  &<\text{bin}> ::= 0 \mid 1
  \end{align*}
  \]

- Goal: Build parse tree for \(1 \ast 1 + 0\) as an \(<\text{exp}>\)
Example cont.

- \[ 1 \times 1 + 0: \quad <\text{exp}> \]

\(<\text{exp}>\) is the start symbol for this parse tree
Example cont.

1 * 1 + 0:  \[<\text{exp}>\]
|                                    
|\[<\text{factor}>\]

Use rule: \(<\text{exp}>::= <\text{factor}>\)
Example cont.

1 * 1 + 0:

Use rule: \(<factor> ::= <bin> * <exp>\)
Example cont.

- $1 \times 1 + 0$:

  \[
  \begin{array}{c}
  \text{<exp>}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{<factor>}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{<bin>}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \times
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{<exp>}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{<factor>}
  \end{array}
  \]

  \[
  \begin{array}{c}
  +
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{<factor>}
  \end{array}
  \]

Use rules: $\text{<bin>} ::= 1$ and $\text{<exp>} ::= \text{<factor>} + \text{<factor>}$
Example cont.

- $1 \times 1 + 0$:

```
  <exp>
    <factor>
      <bin>
        1
      *          <exp>
        <factor> + <factor>
        <bin>    <bin>
```

Use rule: `<factor> ::= <bin>`
Example cont.

- 1 * 1 + 0:

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

Use rules:  

```
<bin> ::= 1 | 0
```
Example cont.

- $1 \times 1 + 0$:

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

Use rules:  
<bin> ::= 1 | 0