Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated
by Vikram Adve and Gul Agha

1. New Programming Paradigm
2. Language Translation
3. Language Semantics

Three Main Topics of the Course

Major Phases of a Compiler

- Source Program
- Lex
- Tokens
- Parse
- Abstract Syntax
- Semantic Analysis
- Symbol Table
- Translate
- Intermediate Representation
- Optimized IR
- Optimized Machine-Specific Assembly Language
- Assembly Language
- Emit code
- Assembler
- Relocatable Object Code
- Linker
- Machine Code

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</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)

Expressions

- if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions
  - `typexpr, -> typexpr2`
- Declarations (in functional languages)
  - `let pattern = expr`
- Statements (in imperative languages)
  - `a = b + c`
- Subprograms
  - `let pattern, = expr, 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

- Start with a given character set – a, b, c...
- $L(\varepsilon) = \{"\"\}$
- Each character is a regular expression
  - It represents the set of one string containing just that character
  - $L(a) = \{a\}$

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 $L(x) = \{a,ab\}$ and $L(y) = \{c,d\}$
  - then $L(xy) = \{ac,ad,abc,abd\}$

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 $L(x^*) = \{\"\",a,ab,aa,aab,abab,...\}$
- $\varepsilon$
  - It represents {""}, set containing the empty string
- $\Phi$
  - 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.

Right Regular Grammars

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

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

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

- 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
  \[ X ::= yZw \]
  \[ Z ::= v \]
  we may replace \( Z \) by \( v \) to say
  \[ X => yZw => yvw \]
- Sequence of such replacements called derivation
- Derivation called 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:
  - \(<Sum> ::= <Sum> + <Sum>\)
  - \(<Sum> => <Sum> + <Sum>\)
Pick a non-terminal:

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

Pick a rule and substitute:

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

Pick a non-terminal:

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

Pick a rule and substitute:

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

Pick a non-terminal:

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

Pick a rule and substitute:

<Sum> ::= 1
<Sum> => <Sum> + <Sum>
  => ( <Sum> ) + <Sum>
  => ( <Sum> + <Sum> ) + <Sum>
  => ( <Sum> + 1 ) + <Sum>
BNF Derivations

Pick a non-terminal:

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

BNF Derivations

Pick a rule and substitute:

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

BNF Derivations

( 0 + 1 ) + 0 is generated by grammar

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

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,...

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 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

- `<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 \(e \lor e\)

**Ocamllex Manual**

More details can be found at


**Example : test.mll**

```ocaml
{ 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 +

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_newline ();
  main newlexbuf }
```

**Example**

```ocaml```
```
# use "test.ml"
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int -> result = <fun>
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**

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

**Example Results**

```
hi there 234 5.2
- : result list = [String "hi"; String "there"; Int 234; Float 5.2]
```

**Dealing with comments**

First Attempt

```ml
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
(digit) '.' digit as f { Float (float_of_string f) :: main lexbuf}
| digit as n   { Int (int_of_string n) :: main lexbuf }
| letter as s  { String s :: main lexbuf}
| eof          { [] }
```

**Your Turn**

- Work on ML5
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers

**Your Turn**

- Work on ML5
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers

**Example**

```
rule main = parse
(digit) '.' digit as f { Float (float_of_string f) :: main lexbuf}
```

**Example Results**

```
hi there 234 5.2
- : result list = [String "hi"; String "there"; Int 234; Float 5.2]
```

**Dealing with comments**

```
First Attempt
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
(digit) '.' digit as f { Float (float_of_string f) :: main lexbuf}
| digit as n   { Int (int_of_string n) :: main lexbuf }
| letter as s  { String s :: main lexbuf}
```

**Used Ctrl-d to send the end-of-file signal**
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 = parse

open_comment { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }
| _ { comment lexbuf }

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 }

and comment depth = parse

open_comment { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }
| _ { comment lexbuf }

Dealing with nested comments

rule main = parse ...

open_comment { comment 1 lexbuf }
| eof              { [] }
| _ { main lexbuf }

and comment = parse

open_comment { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }
| _ { comment lexbuf }

Dealing with nested comments

rule main = parse ...

open_comment { comment 1 lexbuf }
| eof              { [] }
| _ { main lexbuf }

and comment = parse

open_comment { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }
| _ { comment lexbuf }