

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

10/25/19

1

Example

- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

$$f(x) = f(g(f(z), y)) \rightarrow f(g(f(z), f(z))) = f(g(f(z), f(z)))$$

$$g(y, y) = x \rightarrow g(f(z), f(z)) = g(f(z), f(z))$$

10/25/19

2

Example of Failure: Decompose

- Unify $\{(f(x, g(y)) = f(h(y), x))\}$
- Decompose: $(f(x, g(y)) = f(h(y), x))$
- = Unify $\{(x = h(y)), (g(y) = x)\}$
- Orient: $(g(y) = x)$
- = Unify $\{(x = h(y)), (x = g(y))\}$
- Eliminate: $(x = h(y))$
- Unify $\{(h(y) = g(y))\} \circ \{x \rightarrow h(y)\}$
- No rule to apply! Decompose fails!

10/25/19

3

Example of Failure: Occurs Check

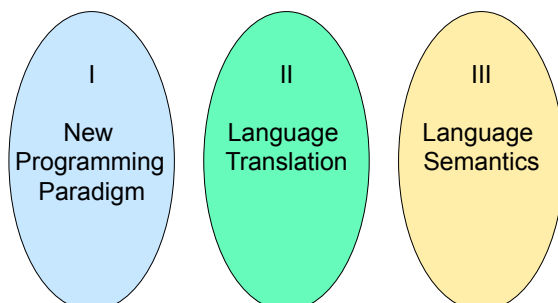
- Unify $\{(f(x, g(x)) = f(h(x), x))\}$
- Decompose: $(f(x, g(x)) = f(h(x), x))$
- = Unify $\{(x = h(x)), (g(x) = x)\}$
- Orient: $(g(x) = x)$
- = Unify $\{(x = h(x)), (x = g(x))\}$
- No rules apply.

10/25/19

4

Programming Languages & Compilers

Three Main Topics of the Course

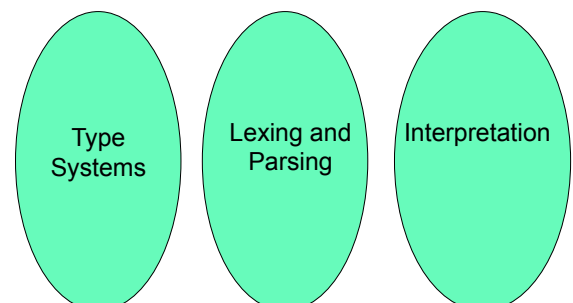


10/25/19

5

Programming Languages & Compilers

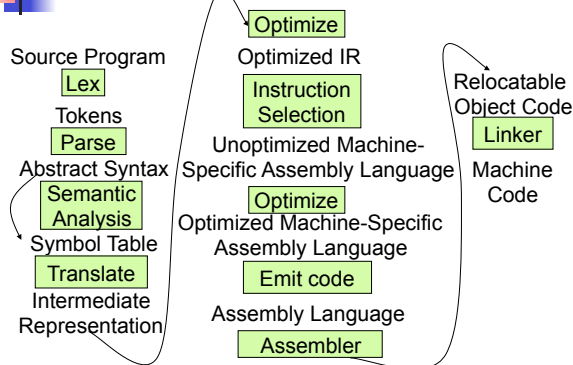
II : Language Translation



10/25/19

6

Major Phases of a Compiler



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)

10/25/19

8

Meta-discourse

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

10/25/19

9

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

10/25/19

10

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

10/25/19

11

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)

10/25/19

12

Elements of Syntax

- Expressions
if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions
 $typexpr_1 \rightarrow typexpr_2$
- Declarations (in functional languages)
let *pattern* = *expr*
- Statements (in imperative languages)
a = b + c
- Subprograms
let *pattern*₁ = *expr*₁ in *expr*

10/25/19

13

Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)

10/25/19

14

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

10/25/19

15

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

10/25/19

16

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

10/25/19

17

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
 - $L(a) = \{a\}$

10/25/19

18

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 $L(xy) = \{ac, ad, abc, abd\}$

10/25/19

19

Regular Expressions

- If **x** and **y** are regular expressions, then **xvy** 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 $L(x \vee y) = \{a, ab, c, d\}$

10/25/19

20

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 $L(x^*) = \{\epsilon, a, ab, aa, aab, abab, \dots\}$
- ϵ
 - It represents $\{\epsilon\}$, set containing the empty string
 - \emptyset
 - It represents $\{\}$, the empty set

10/25/19

21

Example Regular Expressions

- **(0v1)*1**
 - The set of all strings of **0**'s and **1**'s ending in **1**, $\{1, 01, 11, \dots\}$
- **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

10/25/19

22

Regular Grammars

- Subclass of BNF (covered in detail soon)
- Only rules of form $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle \langle \text{nonterminal} \rangle$ or $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle$ or $\langle \text{nonterminal} \rangle ::= \epsilon$
- 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

10/25/19

23

Example

- Regular grammar:
 - $\langle \text{Balanced} \rangle ::= \epsilon$
 - $\langle \text{Balanced} \rangle ::= 0 \langle \text{OneAndMore} \rangle$
 - $\langle \text{Balanced} \rangle ::= 1 \langle \text{ZeroAndMore} \rangle$
 - $\langle \text{OneAndMore} \rangle ::= 1 \langle \text{Balanced} \rangle$
 - $\langle \text{ZeroAndMore} \rangle ::= 0 \langle \text{Balanced} \rangle$
- Generates even length strings where every initial substring of even length has same number of **0**'s as **1**'s

10/25/19

24

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
 - Identifier = $(a \vee b \vee \dots \vee z \vee A \vee B \vee \dots \vee Z)(a \vee b \vee \dots \vee z \vee A \vee B \vee \dots \vee Z \vee 0 \vee 1 \vee \dots \vee 9)^*$
 - Digit = $(0 \vee 1 \vee \dots \vee 9)$
 - Number = $0 \vee (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^* \vee \sim (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^*$
 - Keywords: if = if, while = while,...

10/25/19

25

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

10/25/19

26

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]

10/25/19

27

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

10/25/19

28

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.

10/25/19

29

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.

10/25/19

30

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`

10/25/19

31

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
  main newlexbuf
  }
```

10/28/19

32

General Input

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

10/25/19

33

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

10/25/19

34

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*

10/25/19

35

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 $e_1 \vee e_2$

10/25/19

36

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 $e_1 \vee \epsilon$

10/25/19

37

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 ident = regexp`
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

10/25/19

38

Ocamllex Manual

- More details can be found at

<http://caml.inria.fr/pub/docs/manual-ocaml/lex yacc.html>

10/25/19

39

Example : test.mll

```
{ 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 +
```

10/25/19

40

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_newline ();
  main newlexbuf }
```

10/25/19

41

Example

```
# #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?!?

10/25/19

42

Example

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

10/25/19

43

Your Turn

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

10/25/19

44

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`

10/25/19

45

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 }
```

10/25/19

46

Example Results

```
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

10/25/19

47

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 }
```

10/25/19

48

Dealing with comments

```
| open_comment { comment lexbuf }
| eof          { [] }
| _ { main lexbuf }
and comment = parse
  close_comment { main lexbuf }
| _            { comment lexbuf }
```

10/25/19

49

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 }
```

10/25/19

50

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 }
```

10/25/19

51

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 }
```

10/25/19

52

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

10/25/19

53

Sample Grammar

- Language: Parenthesized sums of 0's and 1's
- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$

10/25/19

54

BNF Grammars

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

10/25/19

55

BNF Grammars

- BNF rules (aka *productions*) have form
$$\mathbf{X} ::= y$$
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

10/25/19

56

Sample Grammar

- Terminals: 0 1 + ()
- Nonterminals: <Sum>
- Start symbol = <Sum>
- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$
- Can be abbreviated as
$$\langle \text{Sum} \rangle ::= 0 \mid 1 \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \mid (\langle \text{Sum} \rangle)$$

10/25/19

57

BNF Derivations

- Given rules
$$\mathbf{X} ::= y\mathbf{Z}w \text{ and } \mathbf{Z} ::= v$$
we may replace **Z** by *v* to say
$$\mathbf{X} \Rightarrow y\mathbf{Z}w \Rightarrow yvw$$
- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal

10/25/19

58

BNF Derivations

- Start with the start symbol:

$\langle \text{Sum} \rangle \Rightarrow$

10/25/19

59

BNF Derivations

- Pick a non-terminal

$\langle \text{Sum} \rangle \Rightarrow$

10/25/19

60

BNF Derivations

- Pick a rule and substitute:
 - $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$$

10/25/19

61

BNF Derivations

- Pick a non-terminal:

$$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$$

10/25/19

62

BNF Derivations

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

10/25/19

63

BNF Derivations

- Pick a non-terminal:

$$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$$
$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

10/25/19

64

BNF Derivations

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

10/25/19

65

BNF Derivations

- Pick a non-terminal:

$$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$$
$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$
$$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

10/25/19

66

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 1$

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$

10/25/19

67

BNF Derivations

- Pick a non-terminal:

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$

10/25/19

68

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 0$

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$

10/25/19

69

BNF Derivations

- Pick a non-terminal:

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$

10/25/19

70

BNF Derivations

- Pick a rule and substitute

- $\langle \text{Sum} \rangle ::= 0$

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) 0$
 $\Rightarrow (0 + 1) + 0$

10/25/19

71


BNF Derivations

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

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$
 $\Rightarrow (0 + 1) + 0$

10/25/19

72



$\langle \text{Sum} \rangle ::= 0 \mid 1 \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \mid (\langle \text{Sum} \rangle)$
 $\langle \text{Sum} \rangle \Rightarrow$