Quiz 4

Midterm 2 ADT, Second Chance

```
type ‘a option =
| None
| Some of ‘a
```

Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- **Reminder:** 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**)
- Last week we started the first step of parsing, which is **lexing** those input strings into **tokens**
- Today we will finish **lexing** and move on to the rest of **parsing**

Objectives for Today

- **Reminder:** 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**)
- Last week we started the first step of parsing, which is **lexing** those input strings into **tokens**
- Today we will finish **lexing** and move on to the rest of **parsing**

Objectives for Today

- **Reminder:** 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**)
- Last week we started the first step of parsing, which is **lexing** those input strings into **tokens**
- Today we will finish **lexing** and move on to the rest of **parsing**

Questions from last week?

Recap

Example : using generated file

```
# #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"
```

Recap

Example : using generated file

```
# #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?

Recap

Example : using generated file

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

Recall the hidden argument of type lexbuf

Recap

Example : using generated file

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

Recall the hidden argument of type lexbuf

Recap

Example : using generated file

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

Recall the hidden argument of type lexbuf

Recap

Fancy Lexing

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**
- **Downside:** *Not* what you want to sew this together with ocamlyacc (parser generator)
- **Side Benefit:** can add **"state"** into lexing
- **Note:** already used this with the _ case

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**
- **Downside:** *Not* what you want to sew this together with ocamlyacc (parser generator)
- **Side Benefit:** can add **"state"** into lexing
- **Note:** already used this with the _ case

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**
- **Downside:** *Not* what you want to sew this together with ocamlyacc (parser generator)
- **Side Benefit:** can add **"state"** into lexing
- **Note:** already used this with the _ case

Example: Old Version

```
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 }
```

Example: WIP New Version

```
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 }
 | _  { main lexbuf }
```

Example: New Version

```
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

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

Questions so far?

Dealing with Comments (No Nesting)

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
... (* same as last time *)
| open_comment { comment lexbuf }
| eof { [] }
| _ { main lexbuf }
and comment = parse
| close_comment { main lexbuf }
| _  { comment lexbuf }
```

Dealing with Comments (No Nesting)

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
... (* same as last time *)
| open_comment { comment lexbuf }
| eof { [] }
| _ { main lexbuf }
and comment = parse
| close_comment { main lexbuf }
| _  { comment lexbuf }
```

Dealing with Comments (No Nesting)

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
... (* same as last time *)
| open_comment { comment lexbuf }
| eof { [] }
| _ { main lexbuf }
and comment = parse
| close_comment { main lexbuf }
| _  { comment lexbuf }
```

Dealing with Comments (No Nesting)

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
... (* same as last time *)
| open_comment { comment lexbuf }
| eof { [] }
| _ { main lexbuf }
and comment = parse
| close_comment { main lexbuf }
| _  { comment lexbuf }
```

Dealing with Comments (No Nesting)

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
... (* same as last time *)
| open_comment { comment lexbuf }
| eof { [] }
| _ { main lexbuf }
and comment = parse
| close_comment { main lexbuf }
| _  { comment lexbuf }
```

Questions so far?

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
…
| 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
…
| 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
…
| 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
…
| 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
…
| 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
…
| 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
…
| 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 }
```

Note: No Longer Regular!

Often easier to defer non-regular things to the parser generator.

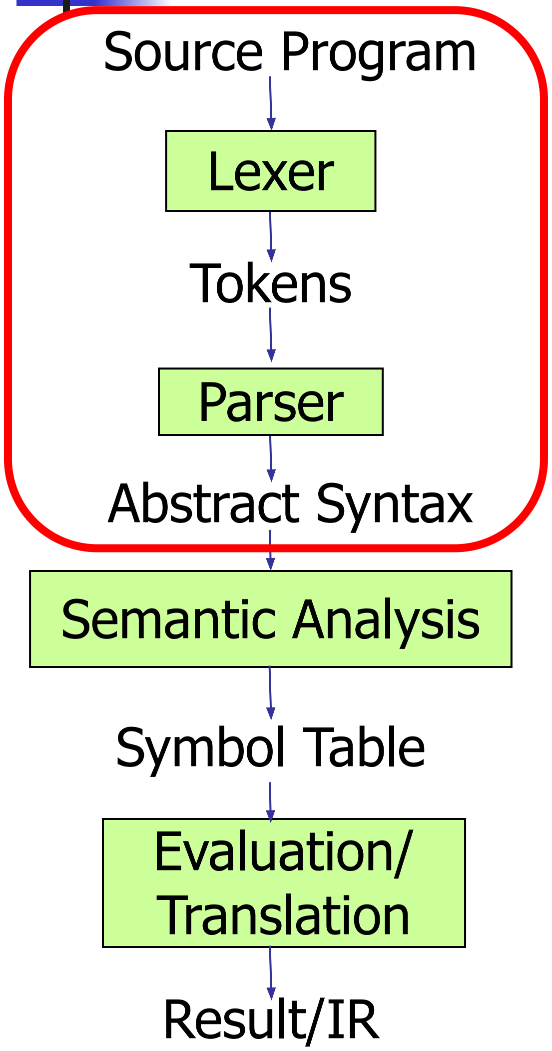
Problem

- How to get lexer to look at **more than the first token** at one time?
- **Answer:** ***action*** has to tell it to – **recursive calls**
- **Downside:** *Not* what you want to sew this together with ocamlyacc (parser generator)
- **Side Benefit:** can add **"state"** into lexing
- **Note:** already used this with the _ case

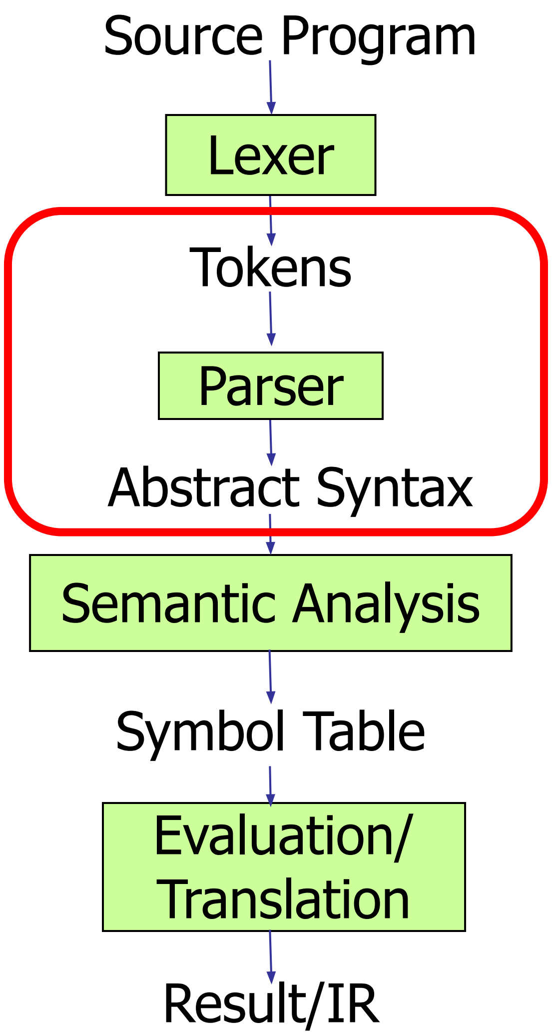
Questions so far?

Parsing

Lexing and Parsing

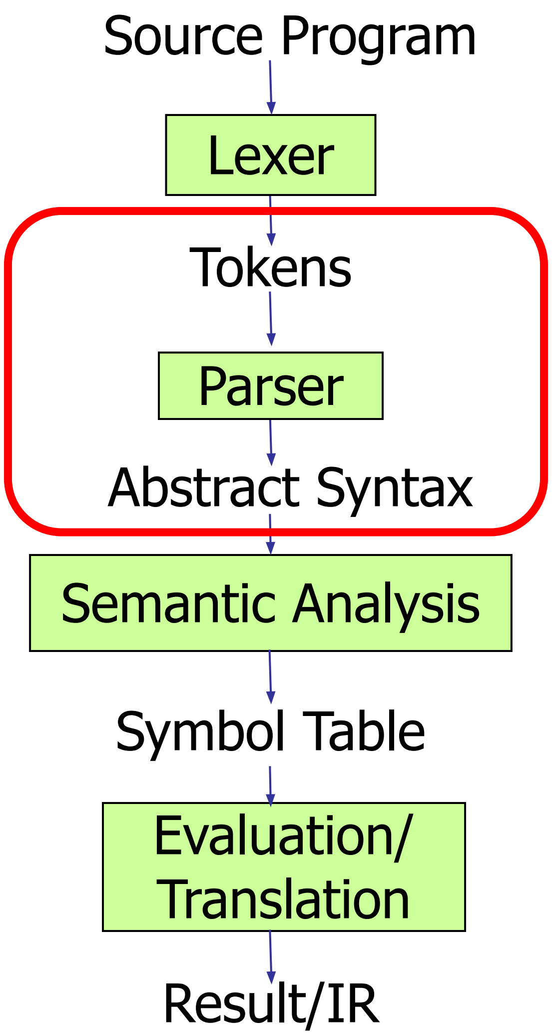


Lexing and **Parsing**



To **parse** our source program and get **abstract syntax**, we need a **grammar** defined in terms of the kinds of **tokens** we get out of our lexer.

Lexing and **Parsing**

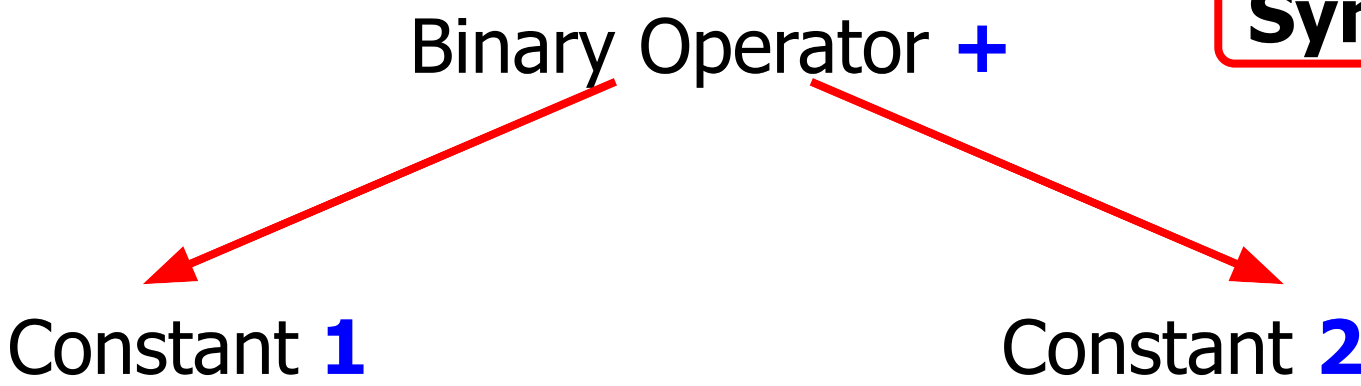


To **parse** our source program and get **abstract syntax**, we need a **grammar** defined in terms of the kinds of **tokens** we get out of our lexer.

The output, an **abstract syntax tree**, will track not just categories, but also **structure**.

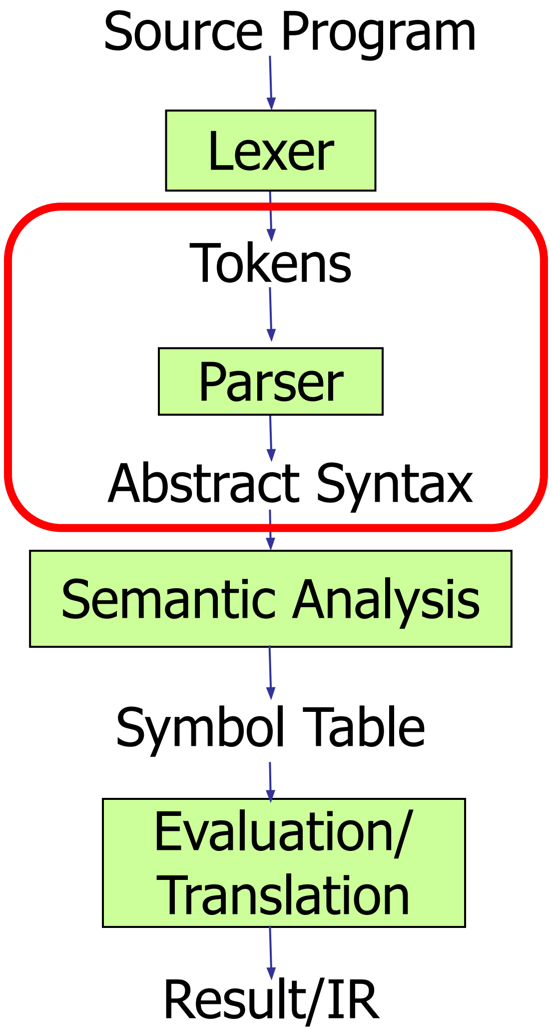
Lexing and **Parsing**

Syntax



The output, an **abstract syntax tree**, will track not just categories, but also **structure**.

Lexing and **Parsing**



To **parse** our source program and get **abstract syntax**, we need a **grammar** defined in terms of the kinds of **tokens** we get out of our lexer.

The output, an **abstract syntax tree**, will track not just categories, but also **structure**.

Sample Grammar

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

<Sum> ::= 0

<Sum> ::= 1

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

<Sum> ::= (<Sum>)

Sample Grammar

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

<Sum> ::= 0

<Sum> ::= 1

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

<Sum> ::= (<Sum>)

Context-Free Grammars

BNF Grammars

- A notation for a **context-free grammar**
- Start with a set of characters **a, b, … (terminals)**
- Add different characters **X, Y, … (nonterminals)**
- One special **nonterminal S** called **start symbol**
- 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

BNF Grammars

- A notation for a **context-free grammar**
- Start with a set of characters **a, b, …** **(terminals)**
- Add different characters **X, Y, …** **(nonterminals)**
- One special **nonterminal** **S** called **start symbol**
- 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

BNF Grammars

- A notation for a **context-free grammar**
- Start with a set of characters **a, b, …** **(terminals)**
- Add different characters **X, Y, …** **(nonterminals)**
- One special **nonterminal** **S** called **start symbol**
- 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

BNF Grammars

- A notation for a **context-free grammar**
- Start with a set of characters **a, b, …** **(terminals)**
- Add different characters **X, Y, …** **(nonterminals)**
- One special **nonterminal** **S** called **start symbol**
- 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

- Terminals: 0 1 + ( )
- Nonterminals: <Sum>
- Start symbol = <Sum>

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

Sample BNF 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>)

Questions so far?

BNF Semantics

- **Question**: What does a BNF grammar **mean**?
- **Answer:** The **meaning** of a BNF grammar is the **set of all strings** consisting only of **terminals** that can be derived from the **Start** symbol
- **Question:** How do we determine that set?

BNF Semantics

- **Question**: What does a BNF grammar **mean**?
- **Answer:** The **meaning** of a BNF grammar is the **set of all strings** consisting only of **terminals** that can be derived from the **Start** symbol
- **Question:** How do we determine that set?

BNF Deriviations

- Given rules

  X ::= y**Z**w and **Z** ::= v

  we may replace **Z** by v to say

  X => y**Z**w => yvw

- Sequence of such replacements called **derivation**
- Derivation called **right-most** if always replace the right-most non-terminal

BNF Deriviations

- Given rules

  X ::= y**Z**w and **Z** ::= v

  we may replace **Z** by v to say

  X => y**Z**w => 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 >

BNF Derivations

Pick a non-terminal:

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

BNF Derivations

Pick a rule and substitute:

- <Sum> ::= (<Sum>)

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

=> (<Sum>) + <Sum>

BNF Derivations

Pick a non-terminal:

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

=> (<Sum>) + <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:

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

=> (<Sum>) + <Sum>

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

BNF Derivations

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

Pick a non-terminal:

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

BNF Derivations

Pick a rule and substitute

- <Sum> ::= 0

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

=> (<Sum>) + <Sum>

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

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

=> (<Sum> + 1) 0

=> (0 + 1) + 0

BNF Derivations

(0 + 1) + 0 is generated by the grammar.

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

=> (<Sum>) + <Sum>

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

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

=> (<Sum> + 1) + 0

=> (0 + 1) + 0

Questions so far?

Extended BNF Grammars

- **Alternatives:** allow rules of from X ::= y | z
  - Abbreviates X ::= y, X ::= z
- **Options:** X ::= y[v]z
  - Abbreviates X ::= yvz, X ::= yz
- **Repetition:** X ::= y{v}*z
  - Can be eliminated by adding new nonterminal V and rules X ::= yz, X ::= yVz, V ::= v, V ::= vV

Extended BNF Grammars

- **Alternatives:** allow rules of from X ::= y | z
  - Abbreviates X ::= y, X ::= z
- **Options:** X ::= y[v]z
  - Abbreviates X ::= yvz, X ::= yz
- **Repetition:** X ::= y{v}*z
  - Can be eliminated by adding new nonterminal V and rules X ::= yz, X ::= yVz, V ::= v, V ::= vV

Extended BNF Grammars

- **Alternatives:** allow rules of from X ::= y | z
  - Abbreviates X ::= y, X ::= z
- **Options:** X ::= y[v]z
  - Abbreviates X ::= yvz, X ::= yz
- **Repetition:** X ::= y{v}*z
  - Can be eliminated by adding new nonterminal V and rules X ::= yz, X ::= yVz, V ::= v, V ::= vV

Questions?

Next Class: From Tokens to ASTs

Next Class

- **EC2 is up**
- **WA7** due **Thursday**
- **MP8** due next **Tuesday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help