

# **Abstract Syntax and Parsing**

CS 350

---

Dr. Joseph Eremondi

Last updated: July 7, 2024

# The Big Picture

---

# Life of a program

- The Language Pipeline:



- Can have many syntaxes that parse to the same abstract syntax

- Can have many syntaxes that parse to the same abstract syntax
  - Different keywords

- Can have many syntaxes that parse to the same abstract syntax
  - Different keywords
  - Different operator names

- Can have many syntaxes that parse to the same abstract syntax
  - Different keywords
  - Different operator names
  - Different order of expressions

- Can have many syntaxes that parse to the same abstract syntax
  - Different keywords
  - Different operator names
  - Different order of expressions
- E.g. plait vs shplait



# **Describing Syntax**

---

- Extended Backus-Naur form

- Extended Backus-Naur form
  - Named after scientists who worked on Algol

- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars

- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars
  - See CS 411

- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars
  - See CS 411
- Describes which strings are valid expressions/statements/etc. in a language

- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars
  - See CS 411
- Describes which strings are valid expressions/statements/etc. in a language
- Generative

- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars
  - See CS 411
- Describes which strings are valid expressions/statements/etc. in a language
- Generative
  - Gives a process for generating valid strings in the language



- Extended Backus-Naur form
  - Named after scientists who worked on Algol
- Notation for Context Free Grammars
  - See CS 411
- Describes which strings are valid expressions/statements/etc. in a language
- Generative
  - Gives a process for generating valid strings in the language
  - String is valid if and only if it's generated by the grammar

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule
- ::= means *can be one of*

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule
- ::= means *can be one of*
- | separates the possibilities

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule
- ::= means *can be one of*
- | separates the possibilities
- Literal strings are in quotation marks

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule
- ::= means *can be one of*
- | separates the possibilities
- Literal strings are in quotation marks
  - Usually for keywords, operators or parentheses

## Example

```
<expr> ::=  
    "{" "+" <expr> <expr> "  
    | "{" "*" <expr> <expr> "  
    | number
```

- <expr> is a *nonterminal*
  - A symbolic variable that doesn't show up in the final string, but is replaced using a rule
- ::= means *can be one of*
- | separates the possibilities
- Literal strings are in quotation marks
  - Usually for keywords, operators or parentheses
- number is a literal number e.g. some sequence of digits



## Generating a string

- Start with a single non-terminal

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression

## Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>}` `-> {+ 2 <expr>}`  
`-> {+ 2 5}`



# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
    `-> {+ 2 5}`
  - `<expr>`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
`-> {+ 2 5}`
  - `<expr>`
    - `-> {* <expr> <expr>}`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
`-> {+ 2 5}`
  - `<expr>`
    - `-> {* <expr> <expr>}`
    - `-> {* {+ <expr> <expr>} <expr>}`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
`-> {+ 2 5}`
  - `<expr>`
    - `-> {* <expr> <expr>}`
    - `-> {* {+ <expr> <expr>} <expr>}`
    - `-> {* {+ 5 <expr>} <expr>}`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
`-> {+ 2 5}`
  - `<expr>`
    - `-> { * <expr> <expr> }`
    - `-> { * {+ <expr> <expr>} <expr> }`
    - `-> { * {+ 5 <expr>} <expr> }`
    - `-> { * {+ 5 100000} <expr> }`

# Generating a string

- Start with a single non-terminal
  - e.g. `<expr>` for an expression
- Until you have a string with no non-terminals, repeatedly:
  - Replace a non-terminal with one of its variants
    - i.e. one of the things on the right of `::=`
- Examples:
  - `<expr> -> 3`
  - `<expr> -> {+ <expr> <expr>} -> {+ 2 <expr>}`  
`-> {+ 2 5}`
  - `<expr>`
    - `-> { * <expr> <expr> }`
    - `-> { * {+ <expr> <expr>} <expr> }`
    - `-> { * {+ 5 <expr>} <expr> }`
    - `-> { * {+ 5 100000} <expr> }`
    - `-> { * {+ 5 100000} -3 }`

# **Parsing and Abstract Syntax**

---

- Notice that the different replacements didn't affect each other



# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node
  - Terminal is a leaf

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node
  - Terminal is a leaf
  - Edge is application of rule from grammar

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node
  - Terminal is a leaf
  - Edge is application of rule from grammar
- Can make a datatype representing these trees

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node
  - Terminal is a leaf
  - Edge is application of rule from grammar
- Can make a datatype representing these trees

# Parse Trees

- Notice that the different replacements didn't affect each other
  - Can effectively replace them in parallel
- Tree structure
  - Non-terminal is a node
  - Terminal is a leaf
  - Edge is application of rule from grammar
- Can make a datatype representing these trees

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```





# Abstract Syntax

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- This is called the *abstract syntax* for the programming language

# Abstract Syntax

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- This is called the *abstract syntax* for the programming language
- A value of this type is called an *abstract syntax tree*

# Abstract Syntax

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- This is called the *abstract syntax* for the programming language
- A value of this type is called an *abstract syntax tree*
  - AST for short

- The process of turning source code (linear string) into abstract syntax (tree)

- The process of turning source code (linear string) into abstract syntax (tree)
  - Turns the program into a thing we process recursively

- The process of turning source code (linear string) into abstract syntax (tree)
  - Turns the program into a thing we process recursively
  - Tree structure mirrors structure of the program

- The process of turning source code (linear string) into abstract syntax (tree)
  - Turns the program into a thing we process recursively
  - Tree structure mirrors structure of the program
- Can fail



- The process of turning source code (linear string) into abstract syntax (tree)
  - Turns the program into a thing we process recursively
  - Tree structure mirrors structure of the program
- Can fail
  - What if the string isn't generated by the grammar?

# Parsing in this class

- Parsing is an interesting problem

## Parsing in this class

- Parsing is an interesting problem
- But it's not an interesting *programming languages* problem

# Parsing in this class

- Parsing is an interesting problem
- But it's not an interesting *programming languages* problem
- We will use Racket/plait features to do most of the parsing for us

- Parsing is an interesting problem
- But it's not an interesting *programming languages* problem
- We will use Racket/plait features to do most of the parsing for us
  - Use quoting to write s-expressions directly

- Parsing is an interesting problem
- But it's not an interesting *programming languages* problem
- We will use Racket/plait features to do most of the parsing for us
  - Use quoting to write s-expressions directly
    - Does the hard work of figuring out nested brackets

# S-expressions

- Symbolic expressions

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT



# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions
- The backtick ' in Racket says “interpret the next thing as an s-exp”

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions
- The backtick `'` in Racket says “interpret the next thing as an s-exp”
  - Single-quote `'` does the same but doesn't handle literals, just symbols

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions
- The backtick `'` in Racket says “interpret the next thing as an s-exp”
  - Single-quote `'` does the same but doesn't handle literals, just symbols



# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions
- The backtick ' in Racket says “interpret the next thing as an s-exp”
  - Single-quote ' does the same but doesn't handle literals, just symbols

```
`a
`(+ 2 3)
`(a b c d (+ 2 3) #t (#f #f))
```

- We'll use backtick as the first half of our parser

# S-expressions

- Symbolic expressions
  - Goes back to John McCarthy, LISP, early days of AI at MIT
  - s-exp for short
- An S-expression is either
  - A symbol
  - A literal (number, boolean, string, etc.)
  - A bracketed list of S-expressions
- The backtick ' in Racket says “interpret the next thing as an s-exp”
  - Single-quote ' does the same but doesn't handle literals, just symbols

```
`a
`(+ 2 3)
`(a b c d (+ 2 3) #t (#f #f))
```

- We'll use backtick as the first half of our parser
  - Easier to deal with s-expressions than strings

## From S-exp to AST

- S-expression is a tree

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list



# From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list
  - If it's an operation, check that we have the right number of arguments

# From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list
  - If it's an operation, check that we have the right number of arguments
  - If we do, (try to) parse each argument

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list
  - If it's an operation, check that we have the right number of arguments
  - If we do, (try to) parse each argument
- Otherwise, fail

## From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list
  - If it's an operation, check that we have the right number of arguments
  - If we do, (try to) parse each argument
- Otherwise, fail
- Uses the `s-exp-match?` function

# From S-exp to AST

- S-expression is a tree
  - Might not be a tree representing anything in our language
- If s-exp is literal
  - Generate the literal in our AST
  - Error if it's unsupported
- If it's a list
  - Check the first thing in the list
  - If it's an operation, check that we have the right number of arguments
  - If we do, (try to) parse each argument
- Otherwise, fail
- Uses the `s-exp-match?` function
  - Don't need to memorize how it works, we'll give you the parsers for the most part