University of Waterloo

Lexing and Parsing

Jon Eyolfson

May 31, 2013

Goal

Formulate a grammar for integer arithmetic

Backus-Naur Form

BNF is a common way we formulate grammars

Given as $G = (T, N, S, P)$, where:

G is the grammar

T is a set of terminals

N is a set of non-terminals

S is the starting non-terminal

P is a set of productions

Terminals

A terminal is usually a token from the lexer

A token is a group of characters with a single meaning

> The lexer is responsible for enforcing these groupings and stripping out white space

Tokens

There are two types of tokens: simple and compound

A simple token is usually a single character

A compound token is defined by a regular expression

Note that we can query the string which matched a token

Our Tokens

Simple: $+, -, *, /$

Compound: INT

Therefore, $T = \{+, -, *, /, \text{INT}\}$

Start Lexing

Regular Expression Operations

Recall a regular expression is made up of characters and the following operators:

·	sequence (implied)
[]	character sets
|	alternation (left or right)
()	grouping
*	repetition (zero or more)
+	repetition (one or more)
?	repetition (zero or one)

The repetition operators apply to the preceding element

Regular Expression Output

When we apply a regular expression to a string and it will match or not

Regular Expression	**String**	**Match**
aa*		No
a+	a	Yes
a+	aa	Yes
(ab)|b	a	No
a?b	b	Yes
a?b	ab	Yes

Formulating an Integer

Recall, we need to define the INT token (or terminal) for our grammar

We would verbally describe it as one or more digits

Regular Expression for Integers

We would use the character set $[0-9]$ to match a single digit

Since we want one or more digits, we can apply + to a digit

Giving us the regular expression $[0-9]+$

Regular Expression Application

How do regular expression implementations actually work?

One way is to convert the regular expression to a finite state automaton (FSA) and apply the input character by character

Finite State Automaton

A FSA contains the following:

- A set of states and state transitions
- A initial state and one or more accepting states

States are arbitrarily numbered and state transitions are for individual characters

FSA Notation

State *n* Initial state Accepting state

State transitions are represented by labelled arrows

FSA Usage

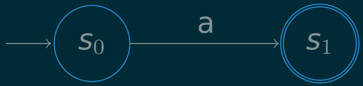
To match a string with our regular expression, do the following:

1. Begin at the initial state
2. Take the state transition matching the current character
   - If no transition, string does not match
3. Repeat step 2 for the next character until there are no more
4. Check if we're in an accepting state
   - If so, the string does match
   - Otherwise, string does not match

FSA Usage Question

Consider the regular expression a

This corresponds to the following FSA:

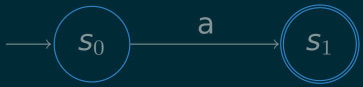


Apply it to the following strings:

1. a
2. ε (fancy way of saying an empty string)
3. bob

Which strings match, and which do not?

FSA Usage Answer



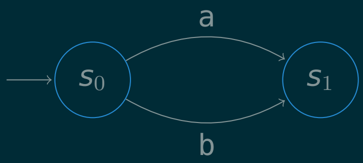
1. a
   - Begin at state s_0
   - Transition to s_1 with character a
   - No more characters, and in an accepting state
   - **Match**
2. ε
   - Begin at state s_0
   - No more characters, and not in an accepting state
   - **No match**
3. bob
   - Begin at state s_0
   - No transition with character b
   - **No match**

FSA Conversions (1)

Regular Expression: ab

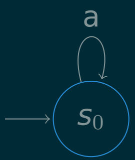
FSA Conversions (2)

Regular Expression: a|b



FSA Conversions (3)

Regular Expression: a*

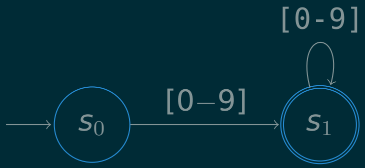


FSA Question

What is a FSA for Integers?

Recall our regular expression: $[0-9]+$

FSA for Integers



Try it with your own input strings

Lexing Question

Consider the following string:

" 1 + 23 * 4 "

With our tokens we defined previously, what does the lexer output?

Lexing Answer

For the following string:

" 1 + 23 * 4 "

Our lexer produces the tokens: INT $+$ INT $*$ INT

End Lexing

Start Language Theory

Regular Languages

All regular expressions can be converted to a FSA

Any language which can be expressed using a FSA is a regular language

Regular Language Limitations

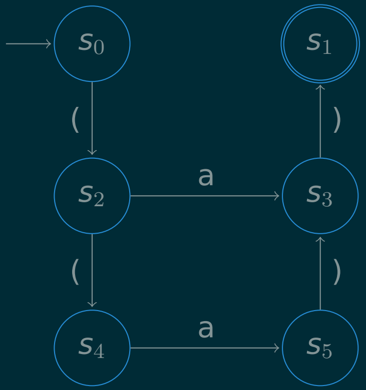
Regular languages cannot handle the following:

- Nesting
- Indefinite counting
- Balancing of symbols

Regular Language Limitations Question

Write a FSA for $(^n\text{a})^n$ for $n \geq 1$ (simple parenthesis matching)

Regular Language Limitations Answer



This is the best we can get in this amount of space (works for $1 \leq n \leq 2$)

The FSA for every value of n needs an infinite size **(contradiction)**

Start Aside

Practical Regular Expression Question

Most regular expression implementations can handle more than the definition of a regular language (we'll consider Perl)

Can you write a regular expression to match: a^n b^n?

Practical Regular Expression Answer

Perl Regular Expression: ^(a(?1)?b)$

This solution uses recursion, (?1) is a reference to the outer group

This expands to the following:

^(a(?1)?b)$
^(a(a(?1)?b)?b)$
etc ...

Source: `http://tinyurl.com/6rayj5a`

End Aside

Continue Language Theory

Push Down Automata

We can change our approach to be able to match $(^n\ a)^n$

We add a stack (a push down stack) and modify our FSA as follows:

- Add a transition condition to optionally check the top of the stack
- Allow transitions to push and pop from the stack

This modified FSA is called a finite state control

The stack and the FSC together form a push down automata

Push Down Automata Notation

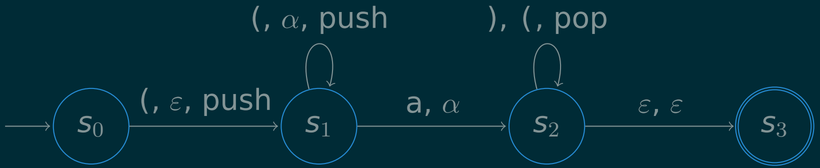
Our transitions are now: character, top of stack, optional push/pop

The character can have a special symbol ε meaning you can take a transition without a character

The top of the stack has two special symbols:

- ε means the top of the stack is empty
- α means the top of the stack may be anything

Push Down Automata Usage Question

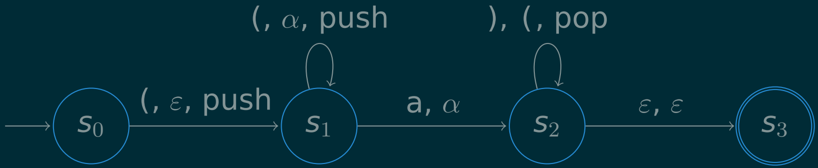


Apply it to the following strings:

1. (a)
2. (a))
3. ((a))
4. ((a)

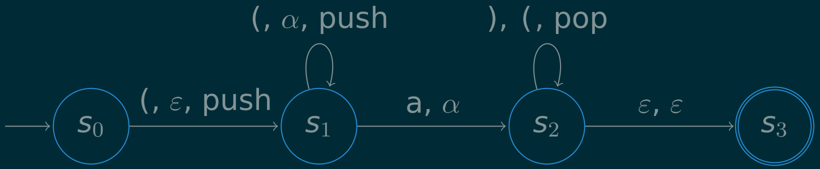
Which strings match, and which do not?

Push Down Automata Usage Answer (1)



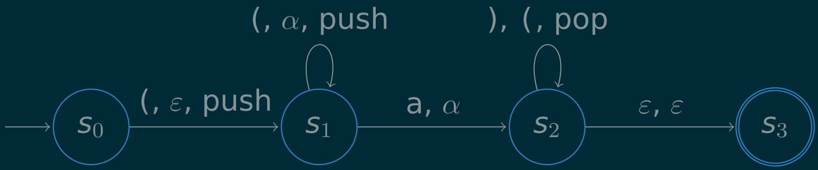
1. (a)
   - Begin at s_0
   - Transition to s_1 with character (and push, stack [(]
   - Transition to s_2 with character a
   - Transition to s_2 with character) and pop, stack []
   - Transition to s_3 since the stack is empty
   - No more characters and in an accepting state
   - **Match**

Push Down Automata Usage Answer (2)



2. (a))
   - Begin at s_0
   - Transition to s_1 with character (and push, stack [(]
   - Transition to s_2 with character a
   - Transition to s_2 with character) and pop, stack []
   - No transition with character)
   - **No match**

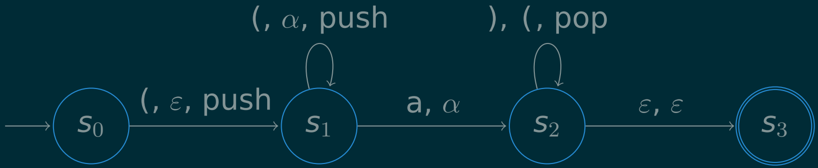
Push Down Automata Usage Answer (3)



3. ((a))
   - Begin at s_0
   - Transition to s_1 with character (and push, stack [(]
   - Transition to s_1 with character (and push, stack [(, (]
   - Transition to s_2 with character a
   - Transition to s_2 with character) and pop, stack [(]
   - Transition to s_2 with character) and pop, stack []
   - Transition to s_3 since the stack is empty
   - No more characters and in an accepting state
   - **Match**

Note that theoretically there is no stack limit, so this works for $n \geq 1$

Push Down Automata Usage Answer (4)



4. ((a)
   - Begin at s_0
   - Transition to s_1 with character (and push, stack [(]
   - Transition to s_1 with character (and push, stack [(, (]
   - Transition to s_2 with character a
   - Transition to s_2 with character) and pop, stack [(]
   - No more characters and not in an accepting state
   - **No match**

Context-Free Languages

Any language which can be expressed using a push down automata or context-free grammar is a context-free language

A more common way to specify a context-free language is to use Backus-Naur Form (BNF)

End Language Theory

Start Parsing

Our Grammar (Attempt 1)

Reminder, our grammar should be in BNF

Given as $G = (T, N, S, P)$, where:

G is the grammar

T is a set of terminals

N is a set of non-terminals

S is the starting non-terminal

P is a set of productions

So far we have, $T = \{+, -, *, /, \text{INT}\}$

BNF Productions

A production is basically a replacement, you may replace the non-terminal with whatever is to the right of the arrow the arrow

A non-terminal is just a production name

Consider,

$$P = \begin{cases} e \to e + e \\ e \to \text{INT} \end{cases}$$

e is our non-terminal, so $N = \{e\}$

BNF Derivations

Consider x and y such that $x, y \in (N \cup T)*$

In other words, x and y are a sequence of terminals and non-terminals

We say x derives y in one step ($x \Rightarrow y$) if we can apply a single production (in P) to x and get y

We say x derives y ($x \Rightarrow^* y$) if we can apply one or more productions (in P) to x to get y

BNF Single Derivation Question

If we have the following:

$x = e + e$
$y = e + e + e$

Does $x \Rightarrow y$?

BNF Single Derivation Answer

If we have the following:

$x = e + e$
$y = e + e + e$

$e + e \Rightarrow e + e + e$ because we can apply $e \rightarrow e + e$ (in P) to x to get to y

Note that we could replace either e

Purpose of a Grammar

The lexer breaks up the input string into a sequence of tokens

The grammar should be used to match this sequence if it's valid

The sequence is valid if we can derive it from the starting non-terminal

Our Grammar (Attempt 2)

Let's drop multiplication and division for now:

$$T = \{+, -, \text{INT}\}$$

$$N = \{e\}$$

$$S = e$$

$$P = \begin{cases} e \to e + e \\ e \to e - e \\ e \to \text{INT} \end{cases}$$

Language Formal Definition

We have a grammar, G, and we want to know what's in our language, L

We define $L(G)$ as the set of all sequences of terminals that can be derived from the starting non-terminal, S

$$L(G) = \{s \mid S \Rightarrow^{*} s \text{ and } s \in T^{*}\}$$

Note that $L(G)$ is likely an infinite set (all possible valid input strings)

BNF Derivation Question

Consider the string "1 + 2 + 3", is it in $L(G)$?

After the lexer, we have the following tokens: $\text{INT} + \text{INT} + \text{INT}$

BNF Derivation Answer

Yes, we can since we can derive $\mathrm{INT} + \mathrm{INT} + \mathrm{INT}$ with the starting non-terminal

$$\begin{aligned} e &\Rightarrow e + e \\ &\Rightarrow e + e + e \\ &\Rightarrow e + e + \mathrm{INT} \\ &\Rightarrow e + \mathrm{INT} + \mathrm{INT} \\ &\Rightarrow \mathrm{INT} + \mathrm{INT} + \mathrm{INT} \end{aligned}$$

Note that the intermediate stages of our derivation which contain terminals and non-terminals is called a sentential form of G

BNF Leftmost Derivation Question

We can do a leftmost derivation by replacing the leftmost non-terminal in a single step

What does our derivation for INT + INT + INT look like now?

BNF Leftmost Derivation Answer (1)

$$
\begin{aligned}
e &\Rightarrow e + e \\
&\Rightarrow e + e + e \\
&\Rightarrow \mathrm{INT} + e + e \\
&\Rightarrow \mathrm{INT} + \mathrm{INT} + e \\
&\Rightarrow \mathrm{INT} + \mathrm{INT} + \mathrm{INT}
\end{aligned}
$$

Parse Tree

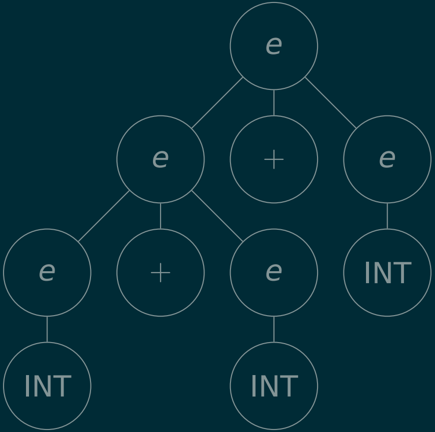
A parse tree is just a visual representation of the derivation

The root node is always the starting non-terminal

The children of any non-terminal is the result of applying the production

What is the parse tree for the previous derivation?

Parse Tree Answer (1)



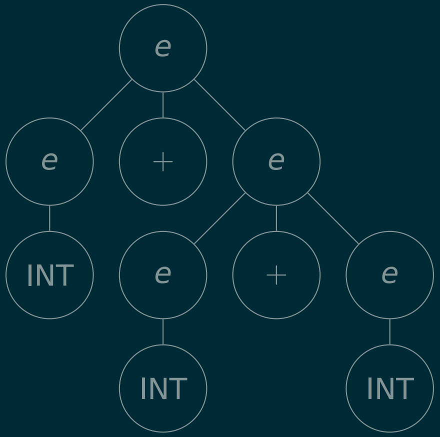
Can we do a different leftmost derivation?

BNF Leftmost Derivaion Answer (2)

$$\begin{aligned} e &\Rightarrow e + e \\ &\Rightarrow \mathrm{INT} + e \\ &\Rightarrow \mathrm{INT} + e + e \\ &\Rightarrow \mathrm{INT} + \mathrm{INT} + e \\ &\Rightarrow \mathrm{INT} + \mathrm{INT} + \mathrm{INT} \end{aligned}$$

And the parse tree?

Parse Tree Answer (2)



Ambiguity

If any of the inputs has more than one leftmost derivation the grammar is called ambiguous

You do not want any ambiguity in your grammar

End Parsing