Introduction to Lexing and Parsing

ECE 351: Compilers

Jon Eyolfson

University of Waterloo

June 18, 2012

Riddle Me This, Riddle Me That

What is a compiler?

Riddle Me This, Riddle Me That

What is a compiler?

It’s a specific kind of language processor.

Terminology

We can think of a language processor as a **black box translator**.

$$\text{Input Language} \longrightarrow \boxed{\text{Magic}} \rightarrow \text{Output Language}$$

Terminology

We can think of a language processor as a **black box translator**.

$$\text{Input Language} \longrightarrow \boxed{\text{Magic}} \rightarrow \text{Output Language}$$

A **compiler** is a translator whose input language is a programming language and outputs machine or assembly language.

More Specific Translators

Assembler

Transliterator (or Preprocessor)

Intermediate Code

- How the input is represented (usually internally) before generating output (e.g. AST, LLVM IR, Bytecode)

Interpreter (or Simulator)

More Specific Translators

Assembler

- Transforms assembly language to machine language

Transliterator (or Preprocessor)

- Transforms one high level language to another

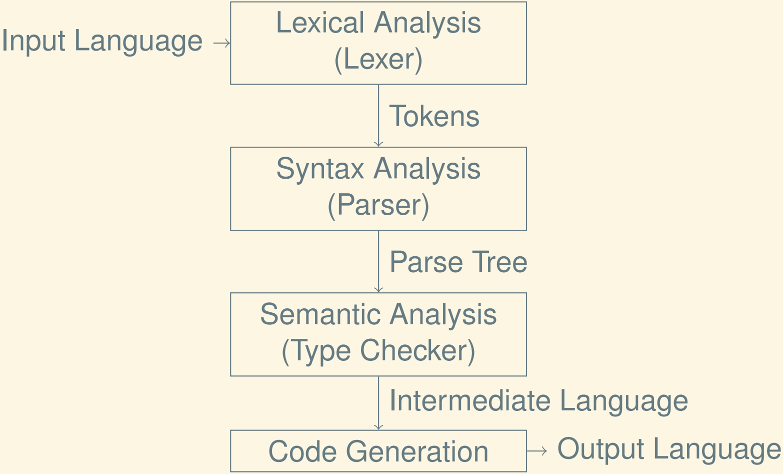
Intermediate Code

- How the input is represented (usually internally) before generating output (e.g. AST, LLVM IR, Bytecode)

Interpreter (or Simulator)

- Directly executes intermediate code

Inside the Black Box



The Lexer

Also known as a **scanner**/**screener**.

Goal

- Break up input characters into groups (tokens)

Why?

- Ignores whitespace
- Provides a nice abstraction

The Lexer

Also known as a **scanner**/**screener**.

Goal

- Break up input characters into groups (tokens)

Why?

- Ignores whitespace
- Provides a nice abstraction

Example

- In `F`, we don't care that the input is "a" or "blahlblahblah", they're both identifiers

Language Definition

Let's revisit some more terminology.

Alphabet - a finite set of symbols

- $\{a, b, c, d, ...\}$

String - any finite sequence of symbols in the alphabet

Empty String - a sequence with no symbols

- ε

Language Definition

Let's revisit some more terminology.

Alphabet - a finite set of symbols

- $\{a, b, c, d, ...\}$

String - any finite sequence of symbols in the alphabet

Empty String - a sequence with no symbols

- ε

Language - a subset of strings in a particular alphabet

Example Language for Identifiers (1)

Alphabet - letters and numbers

- $\{a, b, c, d, ...\}$
- $\{0, 1, 2, 3, ..\}$

Strings

- x
- bbjl15
- 1monaway
- got1

Example Language for Identifiers (1)

Alphabet - letters and numbers

- $\{a, b, c, d, ...\}$
- $\{0, 1, 2, 3, ..\}$

Strings

- x
- bbjl15
- 1monaway
- got1

Which of these strings should belong to our language?

Example Language for Identifiers (1)

Alphabet - letters and numbers

- $\{a, b, c, d, ...\}$
- $\{0, 1, 2, 3, ..\}$

Strings

- x
- bbjl15
- 1monaway
- got1

Which of these strings should belong to our language? Why?

Expressing a Language Using Regular Expressions

Recall a **regular expression** over an alphabet is made up of symbols (in the alphabet) and the following operators:

$*$	repetition (zero or more)
|	alternation (or)
$\cdot$	sequence (implied)
()	grouping
[]	character sets

Expressing a Language Using Regular Expressions

Recall a **regular expression** over an alphabet is made up of symbols (in the alphabet) and the following operators:

*	repetition (zero or more)
|	alternation (or)
·	sequence (implied)
()	grouping
[]	character sets

Notes:

- `a+` $\equiv$ `a·a*` (one or more)
- `a?b` $\equiv$ `b|(a·b)` (zero or one)

Example Language for Identifiers (2)

If our language for identifiers should begin with a letter followed by any number of letters and numbers, what should our regular expression be?

Hint: `[A-Z]`, `[a-z]` and `[0-9]` may be useful.

Example Language for Identifiers (2)

If our language for identifiers should begin with a letter followed by any number of letters and numbers, what should our regular expression be?

Hint: `[A-Z]`, `[a-z]` and `[0-9]` may be useful.

Answer: `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*`

Using Regular Expressions

So, how do we use regular expressions (or what does `grep` do)?

One way is to convert the regular expression to a **finite state automaton** (FSA) and follow it for each input character.

Using Regular Expressions

So, how do we use regular expressions (or what does `grep` do)?

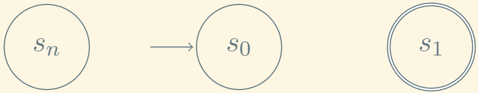
One way is to convert the regular expression to a **finite state automaton** (FSA) and follow it for each input character.

Finite State Automaton

- A set of **states** and **state transitions**
- Contains a **start state** and one or more **final states**

States can be arbitrarily numbered, state transitions are for individual symbols in the alphabet (characters)

Finite State Automaton Notation



State n Starting state Final state

State transitions are represented by labeled arrows

Finite State Automaton Usage

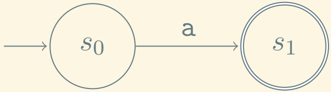
To see if a sentence is in our language we do the following:

1. Start a the starting state(!)
2. Follow the state transition for each character
   - No transition, reject
3. Accept if we're in a final state, reject otherwise

Finite State Automaton Example

Consider the simplest language, represented by the regular expression `a`. Implicitly our alphabet is the set of keyboard characters.

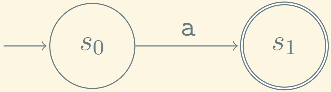
This corresponds to the following FSA:



Finite State Automaton Example

Consider the simplest language, represented by the regular expression `a`. Implicitly our alphabet is the set of keyboard characters.

This corresponds to the following FSA:



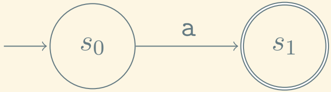
Do we accept or reject these sentences?

- `a`
- ε
- `bob`

Finite State Automaton Example

Consider the simplest language, represented by the regular expression `a`. Implicitly our alphabet is the set of keyboard characters.

This corresponds to the following FSA:



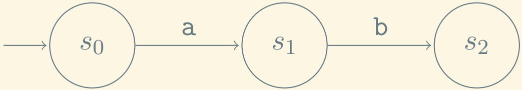
Do we accept or reject these sentences?

- `a`
- ε
- `bob`

Answer: we only accept `a`

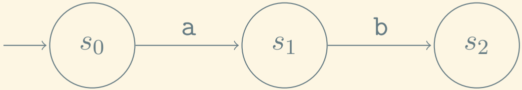
Finite State Automaton Basic Conversions

Regular Expression: `a · b`

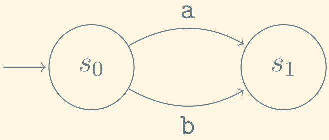


Finite State Automaton Basic Conversions

Regular Expression: `a · b`

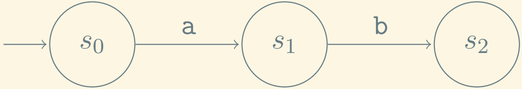


Regular Expression: `a|b`

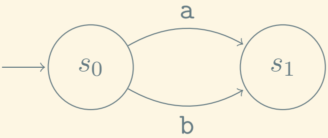


Finite State Automaton Basic Conversions

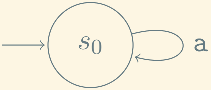
Regular Expression: `a · b`



Regular Expression: `a|b`



Regular Expression: `a*`



Finite State Automaton for Identifiers

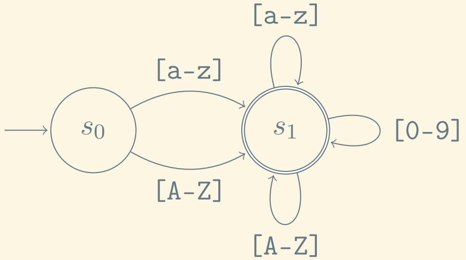
What does the FSA look like for identifiers?

Recall: `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*`

Finite State Automaton for Identifiers

What does the FSA look like for identifiers?

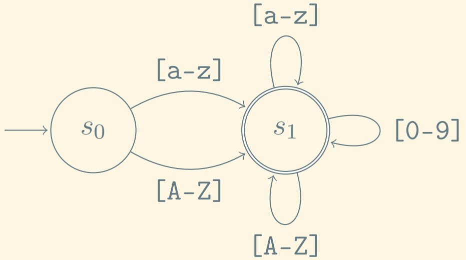
Recall: `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*`



Finite State Automaton for Identifiers

What does the FSA look like for identifiers?

Recall: `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*`



This accepts “bbjl15” and rejects “1monaway”

Regular Languages

It is known all regular expressions can be converted to a FSA

- Any language which can be expressed using a regular expression or a FSA is a **regular language**

Regular Languages

It is known all regular expressions can be converted to a FSA

- Any language which can be expressed using a regular expression or a FSA is a **regular language**

For example, `W` is a regular language, `F` is not. **Why?**

Regular Languages

It is known all regular expressions can be converted to a FSA

- Any language which can be expressed using a regular expression or a FSA is a **regular language**

For example, `W` is a regular language, `F` is not. **Why?**

Regular languages cannot handle:

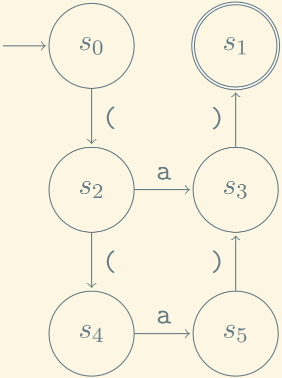
- Nesting
- Indefinite counting
- Balancing of symbols

Illustration of Regular Language Limitations

Can we write a FSA for $\texttt{(}^n\ \texttt{a}\texttt{)}^n$ (simple parenthesis matching)?

Illustration of Regular Language Limitations

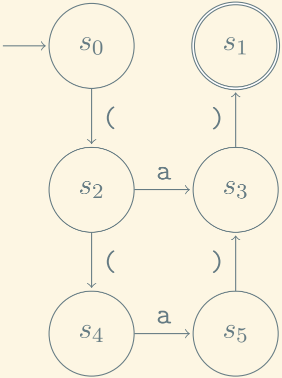
Can we write a FSA for `(`^{n} `a)`^{n} (simple parenthesis matching)?



This is as close as we can get (in this amount of space)

Illustration of Regular Language Limitations

Can we write a FSA for $\texttt{(}^n\ \texttt{a}\texttt{)}^n$ (simple parenthesis matching)?



This is as close as we can get (in this amount of space)

- We need an FSA of infinite size **(contradiction!)**

Real Regular Expressions

While this is technically correct (the best kind of correct) most regular expression implementations are somewhere in the grey area

Can you write a regular expression to match: $\mathtt{a}^n\ \mathtt{b}^n$?

Real Regular Expressions

While this is technically correct (the best kind of correct) most regular expression implementations are somewhere in the grey area

Can you write a regular expression to match: $\mathtt{a}^n\ \mathtt{b}^n$?

With Perl Regular Expressions, we can use: `^(a(?1)?b)$`

Real Regular Expressions

While this is technically correct (the best kind of correct) most regular expression implementations are somewhere in the grey area

Can you write a regular expression to match: $\mathtt{a}^n\ \mathtt{b}^n$?

With Perl Regular Expressions, we can use: ^(a(?1)?b)$

- Basically, (?1) matches (a(?1)?b) and recurses to match the same number of a's and b's

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

This is just for general interest, no need to worry

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

Push Down Automata

We can modify our FSA to be able to match $\texttt{(}^n\ \texttt{a}\texttt{)}^n$ as follows:

- Add a push down stack
- Add another condition for a transition
  1. The input symbol (as before)
  2. The top symbol on the stack
- Allow transitions to push and pop from the stack

Push Down Automata

We can modify our FSA to be able to match $\texttt{(}^n\ \texttt{a)}^n$ as follows:

- Add a push down stack
- Add another condition for a transition
  1. The input symbol (as before)
  2. The top symbol on the stack
- Allow transitions to push and pop from the stack

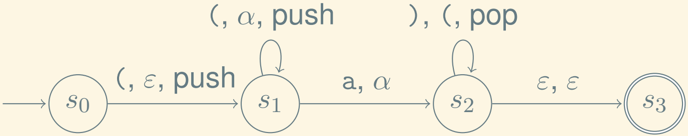
The modified FSA is called a **finite state control**

The stack and the FSC together form a **push down automata**

Push Down Automata Example

Notation:

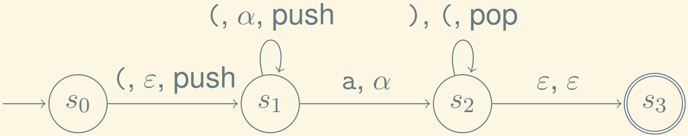
- ε means the top of the stack is empty
- α means the top of the stack may be anything
- Transitions are: symbol, top of stack, optional push/pop



Push Down Automata Example

Notation:

- ε means the top of the stack is empty
- α means the top of the stack may be anything
- Transitions are: symbol, top of stack, optional push/pop



Theoretically there is no stack limit, so this works

Examples: `(a)`, `(((a)))` are accepted and `(a))` is rejected

Context-Free Language

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

We haven't used a push down automata, and neither have any of our tools, how did express a grammar for `F`?

Context-Free Language

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

We haven't used a push down automata, and neither have any of our tools, how did express a grammar for `F`?

We used a context-free grammar, which is specified in **Extended Backus-Naur Form (BNF)**

Backus-Naur Form

BNF is a 4-tuple (T, N, S, P), where

- T is a set of **terminal** symbols (tokens)
- N is a set of **nonterminal** symbols (rule names)
- S is the starting rule, which is a member of N
- P is a set of **rules** (or productions)

Backus-Naur Form

BNF is a 4-tuple (T, N, S, P), where

- T is a set of **terminal** symbols (tokens)
- N is a set of **nonterminal** symbols (rule names)
- S is the starting rule, which is a member of N
- P is a set of **rules** (or productions)

All rules have the form: $A \rightarrow \gamma$

$A \in N$ (A is a nonterminal)
$\gamma \in (N \cup T)*$ (γ is a string of terminals/nonterminals or ε)

Note: $B \rightarrow C|D$ is shorthand for $B \rightarrow C, B \rightarrow D$

Backus-Naur Form Example

Consider the grammar $G = (T, N, S, P)$, where

- $T = \{\, \texttt{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +} \,\}$
- $N = \{\, \texttt{E} \,\}$
- $S = \texttt{E}$
- $P = \texttt{E} \rightarrow \texttt{E + E}$

  $\texttt{E} \rightarrow \texttt{1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0}$

Backus-Naur Form Derivations

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

- x and y are strings of terminals/nonterminals or ε

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

Backus-Naur Form Derivations

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

- x and y are strings of terminals/nonterminals or ε

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

$\texttt{E + E} \Rightarrow \texttt{E + E + E}$ since $\texttt{E} \rightarrow \texttt{E + E} \in P$

Backus-Naur Form Derivations

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

- x and y are strings of terminals/nonterminals or ε

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

$\texttt{E + E} \Rightarrow \texttt{E + E + E}$ since $\texttt{E} \rightarrow \texttt{E + E} \in P$

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

Backus-Naur Form Usage

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

Our strings in this case are a sequence of terminals (or tokens)

Backus-Naur Form Usage

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

Our strings in this case are a sequence of terminals (or tokens)

$L(G)$ is the set of all strings of terminals that can be derived from the starting rule S

In other words (CS): $L(G) = \{s \mid S \Rightarrow^* s$ and $s \in T^*\}$

Note: $L(G)$ is likely an infinite set (all possible valid programs)

Backus-Naur Form Derivation Example

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

Yes, since:

$$
\begin{aligned}
\texttt{E} &\Rightarrow \texttt{E + E} \\
&\Rightarrow \texttt{E + E + E} \\
&\Rightarrow \texttt{E + E + 3} \\
&\Rightarrow \texttt{E + 2 + 3} \\
&\Rightarrow \texttt{1 + 2 + 3}
\end{aligned}
$$

Backus-Naur Form Derivation Example

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

Yes, since:

$$
\begin{aligned}
\texttt{E} &\Rightarrow \texttt{E + E} \\
&\Rightarrow \texttt{E + E + E} \\
&\Rightarrow \texttt{E + E + 3} \\
&\Rightarrow \texttt{E + 2 + 3} \\
&\Rightarrow \texttt{1 + 2 + 3}
\end{aligned}
$$

Note: if our derivation contains terminals and nonterminals, we call it a **sentential form** of G

BNF Leftmost Derivation (1)

Consider the string `1 + 2 + 3` again, we can do a **leftmost derivation** by replacing the leftmost nonterminal in every step

$$
\begin{aligned}
\mathtt{E} &\Rightarrow \mathtt{E + E} \\
&\Rightarrow \mathtt{E + E + E} \\
&\Rightarrow \mathtt{1 + E + E} \\
&\Rightarrow \mathtt{1 + 2 + E} \\
&\Rightarrow \mathtt{1 + 2 + 3}
\end{aligned}
$$

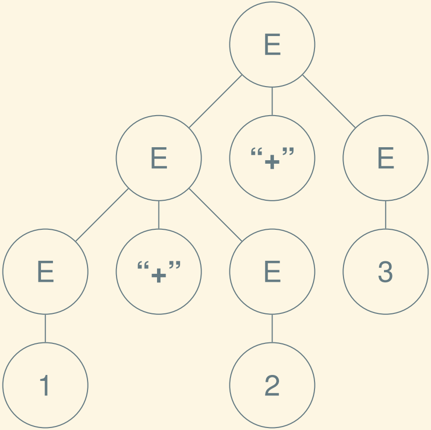
BNF Leftmost Derivation (1)

Consider the string `1 + 2 + 3` again, we can do a **leftmost derivation** by replacing the leftmost nonterminal in every step

$$
\begin{aligned}
\mathtt{E} &\Rightarrow \mathtt{E + E} \\
&\Rightarrow \mathtt{E + E + E} \\
&\Rightarrow \mathtt{1 + E + E} \\
&\Rightarrow \mathtt{1 + 2 + E} \\
&\Rightarrow \mathtt{1 + 2 + 3}
\end{aligned}
$$

This corresponds to the following parse tree...

Parse Tree (1)



BNF Leftmost Derivation (2)

Again, considering `1 + 2 + 3` again, there's another leftmost derivation, what is it?

BNF Leftmost Derivation (2)

Again, considering `1 + 2 + 3` again, there's another leftmost derivation, what is it?

$$
\begin{aligned}
\mathtt{E} &\Rightarrow \mathtt{E + E} \\
&\Rightarrow \mathtt{1 + E} \\
&\Rightarrow \mathtt{1 + E + E} \\
&\Rightarrow \mathtt{1 + 2 + E} \\
&\Rightarrow \mathtt{1 + 2 + 3}
\end{aligned}
$$

If there's more than one leftmost derivation the grammar is **ambiguous** (that's bad)

BNF Leftmost Derivation (2)

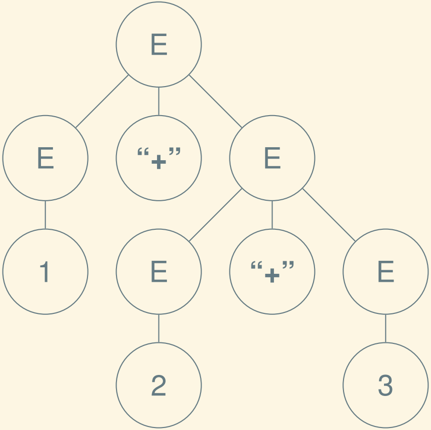
Again, considering `1 + 2 + 3` again, there's another leftmost derivation, what is it?

$$
\begin{aligned}
\texttt{E} &\Rightarrow \texttt{E + E} \\
&\Rightarrow \texttt{1 + E} \\
&\Rightarrow \texttt{1 + E + E} \\
&\Rightarrow \texttt{1 + 2 + E} \\
&\Rightarrow \texttt{1 + 2 + 3}
\end{aligned}
$$

If there's more than one leftmost derivation the grammar is **ambiguous** (that's bad)

This corresponds to the following parse tree...

Parse Tree (2)



Summary

- Definitions for **string**, **language**, **regular language** and **context-free language**
- Creating and using **finite state automaton**
- Using **BNF** grammars and detecting ambiguous grammars

Lab 7

Use `EOI` in the expansion of your starting rule

This makes sure parboiled tries to parse the entire input

Common Problems:

- Your output AST is missing a bunch of input
- You're recognizing strings you shouldn't be

Solution: `Sequence(ZeroOrMore(DesignUnit()), EOI)`

Next Lecture

- Removing ambiguity using precedence and associativity
- Extended Backus-Naur Form (EBNF)
- Other sources of ambiguity
- Methods of parsing