What is a compiler?
What is a compiler?
It’s a specific kind of language processor.
We can think of a language processor as a **black box translator**.

**Input Language** → **Magic** → **Output Language**
We can think of a language processor as a **black box translator**.

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

Input Language →

Lexical Analysis (Lexer) → Tokens

Syntax Analysis (Parser) → Parse Tree

Semantic Analysis (Type Checker) → Intermediate Language

Code Generation → Output Language
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 “blahblahblah”, they’re both identifiers
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
- \(\varepsilon\)
Let’s revisit some more terminology.

**Alphabet** - a finite set of symbols

- \{a, b, c, d, \ldots\}

**String** - any finite sequence of symbols in the alphabet

**Empty String** - a sequence with no symbols

- \(\varepsilon\)

**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, \ldots\}
- \{0, 1, 2, 3, \ldots\}

**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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>repetition (zero or more)</td>
</tr>
<tr>
<td></td>
<td>alternation (or)</td>
</tr>
<tr>
<td>·</td>
<td>sequence (implied)</td>
</tr>
<tr>
<td>(</td>
<td>grouping</td>
</tr>
<tr>
<td>[]</td>
<td>character sets</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>repetition (zero or more)</td>
</tr>
<tr>
<td>|</td>
<td>alternation (or)</td>
</tr>
<tr>
<td>\cdot</td>
<td>sequence (implied)</td>
</tr>
<tr>
<td>()</td>
<td>grouping</td>
</tr>
<tr>
<td>[]</td>
<td>character sets</td>
</tr>
</tbody>
</table>

Notes:

- $a^+ \equiv a \cdot a^*$ (one or more)
- $a?b \equiv b | (a \cdot b)$ (zero or one)
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:** \(( [A-Z] | [a-z] ) ( [A-Z] | [a-z] | [0-9] )^* \)
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 transitions are represented by labeled arrows

State $s_n$  Starting state  Final state $s_1$
Finite State Automaton Usage

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

1. Start at 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:

```
s0 ----> a ----> s1
```
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?

- $a$
- $\varepsilon$
- bob
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?

- $a$
- $\varepsilon$
- bob

**Answer:** we only accept $a$
Finite State Automaton Basic Conversions

Regular Expression: \( a \cdot b \)

\[ \begin{array}{c}
\rightarrow s_0 \quad a \quad s_1 \quad b \quad s_2
\end{array} \]
Finite State Automaton Basic Conversions

Regular Expression: \( a \cdot b \)

\[
\begin{array}{ccc}
  & s_0 & \\
\rightarrow & a & \rightarrow \\
  & s_1 & \rightarrow \ b \\
\end{array}
\]

Regular Expression: \( a | b \)

\[
\begin{array}{ccc}
  & s_0 & \\
\rightarrow & a & \rightarrow \\
  & s_1 & \rightarrow \ b \\
\end{array}
\]
Finite State Automaton Basic Conversions

**Regular Expression:** \( a \cdot b \)

\[
\begin{array}{c}
\text{\( s_0 \)} \\
\rightarrow \\
\text{\( s_1 \)} \\
\text{\( a \)} \\
\rightarrow \\
\text{\( s_2 \)} \\
\text{\( b \)}
\end{array}
\]

**Regular Expression:** \( a | b \)

\[
\begin{array}{c}
\text{\( s_0 \)} \\
\rightarrow \\
\text{\( s_1 \)} \\
\text{\( a \)} \\
\rightarrow \\
\text{\( s_0 \)} \\
\text{\( b \)} \\
\rightarrow \\
\text{\( s_1 \)}
\end{array}
\]

**Regular Expression:** \( a^* \)

\[
\begin{array}{c}
\text{\( s_0 \)} \\
\rightarrow \\
\text{\( s_0 \)} \\
\text{\( a \)}
\end{array}
\]
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"
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, \( \mathcal{L} \) is a regular language, \( \mathcal{F} \) 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, $\mathcal{W}$ is a regular language, $\mathcal{F}$ is not. **Why?**

Regular languages cannot handle:

- Nesting
- Indefinite counting
- Balancing of symbols
Illustration of Regular Language Limitations

Can we write a FSA for \( (n \ a)^n \) (simple parenthesis matching)?

• We need an FSA of infinite size (contradiction!)
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 \((n \ a)^n\) (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: $a^n b^n$?

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(a(?1)?b)?b)$
```

... This is just for general interest, no need to worry

Source: [http://tinyurl.com/6rayj5a](http://tinyurl.com/6rayj5a)
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: $a^n b^n$?

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: $a^n b^n$?

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 \((n \ a)^n\) 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
Push Down Automata

We can modify our FSA to be able to match \((^n a)^n\) 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
Notation:

- \( \varepsilon \) means the top of the stack is empty
- \( \alpha \) 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
Push Down Automata Example

Notation:

- \( \varepsilon \) means the top of the stack is empty
- \( \alpha \) means the top of the stack may be anything
- Transitions are: symbol, top of stack, optional push/pop

There are transitions:

- \((, \varepsilon, \text{push} ), (, \text{pop})\)
- \((, \alpha, \text{push})\)
- \(a, \alpha\)
- \(\varepsilon, \varepsilon\)

Theoretically there is no stack limit, so this works

Examples: \((a), (((a)))\) are accepted and \((a))\) is rejected
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)
$\gamma \in (N \cup T)^*$ ($\gamma$ is a string of terminals/nonterminals or $\epsilon$)

**Note:** $B \rightarrow C|D$ is shorthand for $B \rightarrow C, B \rightarrow D$
Consider the grammar $G = (T, N, S, P)$, where

- $T = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +\}$
- $N = \{E\}$
- $S = E$
- $P = E \rightarrow E + E$
  \[
  E \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0
  \]
Consider $x$ and $y$ such that $x, y \in (N \cup T)^*$

- $x$ and $y$ are strings of terminals/nonterminals or $\varepsilon$

We say $x$ derives $y$ in one step ($x \Rightarrow y$) if we can apply a single rule (in $P$) to $x$ and get $y$
Consider $x$ and $y$ such that $x, y \in (N \cup T)^*$

- $x$ and $y$ are strings of terminals/nonterminals or $\varepsilon$

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

$E + E \Rightarrow E + E + E$ since $E \rightarrow E + E \in P$
Consider $x$ and $y$ such that $x, y \in (N \cup T)^*$

- $x$ and $y$ are strings of terminals/nonterminals or $\varepsilon$

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

$$E + E \Rightarrow E + E + E$$ since $E \rightarrow E + E \in P$

We say $x$ derives $y$ ($x \Rightarrow^* y$) if we can apply one or more steps to $x$ to get $y$
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)
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): $L(G) = \{ s \mid S \Rightarrow^* s \text{ and } s \in T^* \}$

**Note:** $L(G)$ is likely an infinite set (all possible valid programs)
Consider the string $1 + 2 + 3$, is it in $L(G)$?

Yes, since:

\[
E \Rightarrow E + E \\
\Rightarrow E + E + E \\
\Rightarrow E + E + 3 \\
\Rightarrow E + 2 + 3 \\
\Rightarrow 1 + 2 + 3
\]
Consider the string $1 + 2 + 3$, is it in $L(G)$?

Yes, since:

$$E \Rightarrow E + E$$
$$\Rightarrow E + E + E$$
$$\Rightarrow E + E + 3$$
$$\Rightarrow E + 2 + 3$$
$$\Rightarrow 1 + 2 + 3$$

**Note:** if our derivation contains terminals and nonterminals, we call it a *sentential form* of $G$.
Consider the string $1 + 2 + 3$ again, we can do a leftmost derivation by replacing the leftmost nonterminal in every step.

\[
\begin{align*}
E & \Rightarrow E + E \\
& \Rightarrow E + E + E \\
& \Rightarrow 1 + E + E \\
& \Rightarrow 1 + 2 + E \\
& \Rightarrow 1 + 2 + 3
\end{align*}
\]
Consider the string $1 + 2 + 3$ again, we can do a **leftmost derivation** by replacing the leftmost nonterminal in every step.

\[
E \Rightarrow E + E \\
\Rightarrow E + E + E \\
\Rightarrow 1 + E + E \\
\Rightarrow 1 + 2 + E \\
\Rightarrow 1 + 2 + 3
\]

This corresponds to the following parse tree...
Again, considering $1 + 2 + 3$ again, there’s another leftmost derivation, what is it?
Again, considering $1 + 2 + 3$ again, there’s another leftmost derivation, what is it?

$$
E \Rightarrow E + E \\
\Rightarrow 1 + E \\
\Rightarrow 1 + E + E \\
\Rightarrow 1 + 2 + E \\
\Rightarrow 1 + 2 + 3
$$

If there’s more than one leftmost derivation the grammar is **ambiguous** (that’s bad)
Again, considering $1 + 2 + 3$ again, there’s another leftmost derivation, what is it?

$$
\begin{align*}
E & \Rightarrow E + E \\
& \Rightarrow 1 + E \\
& \Rightarrow 1 + E + E \\
& \Rightarrow 1 + 2 + E \\
& \Rightarrow 1 + 2 + 3
\end{align*}
$$

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
Use \texttt{EOI} in the expansion of your starting rule

This makes sure parboiled tries to parse the entire input

\textbf{Common Problems:}
- Your output AST is missing a bunch of input
- You’re recognizing strings you shouldn’t be

\textbf{Solution:} \texttt{Sequence(ZeroOrMore(DesignUnit()), EOI)}
Next Lecture

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