01 - Introduction and Math Preliminaries

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Maryville COLLEGE

Outline

Outline

What is a compiler?

A compiler ...

What is a compiler?

A compiler ...

- verifies the validity of the source program.

What is a compiler?

A compiler ...

- verifies the validity of the source program.
- translate a source program into an object program.

What is a compiler?

A compiler ...

- verifies the validity of the source program.
- translate a source program into an object program.
- translates a source program without changing its semantic meaning.

Program Stages

Program Stages

- Compile Time

Program Stages

- Compile Time
  - Lexicographical Properties of the Program

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time
  - Loading and linking of shared libraries

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time
  - Loading and linking of shared libraries
  - Relocation of Code

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time
  - Loading and linking of shared libraries
  - Relocation of Code
- Run Time

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time
  - Loading and linking of shared libraries
  - Relocation of Code
- Run Time
  - Program Execution

Program Stages

- Compile Time
  - Lexicographical Properties of the Program
  - Validation
  - Code Production
- Load Time
  - Loading and linking of shared libraries
  - Relocation of Code
- Run Time
  - Program Execution
  - Dynamic Behavior of the Program

Phases of Compilation

1. Lexical Analysis

Phases of Compilation

1. Lexical Analysis
2. Syntax Analysis

Phases of Compilation

1. Lexical Analysis
2. Syntax Analysis
3. Code Generation

Lexical Analysis

- Analysis of **microsyntax** of a language.

Lexical Analysis

- Analysis of **microsyntax** of a language.
- Breaking a program into tokens (aka basic symbols or lexemes)

Lexical Analysis

- Analysis of **microsyntax** of a language.
- Breaking a program into tokens (aka basic symbols or lexemes)
- Not always trivial! (Context can alter tokens)
  Consider the following Fortran:

Lexical Analysis

- Analysis of **microsyntax** of a language.
- Breaking a program into tokens (aka basic symbols or lexemes)
- Not always trivial! (Context can alter tokens)
  Consider the following Fortran:
  - `DO 1 I=1,12`

Lexical Analysis

- Analysis of **microsyntax** of a language.
- Breaking a program into tokens (aka basic symbols or lexemes)
- Not always trivial! (Context can alter tokens)
  Consider the following Fortran:
  - `DO 1 I=1,12`
  - `DO 1 I=1.12`

Syntax Analysis

- Analyzes the structure of the program.

Syntax Analysis

- Analyzes the structure of the program.
- Validates structure. (i.e. Do { } match?)

Syntax Analysis

- Analyzes the structure of the program.
- Validates structure. (i.e. Do { } match?)
- Results in a **parse tree** representation of the program.

Code Generation

The code generator ...

- traverses the parse tree.

Code Generation

The code generator ...

- traverses the parse tree.
- generates object code as it descends the tree.

Code Generation

The code generator ...

- traverses the parse tree.
- generates object code as it descends the tree.
- optimizes object code.

Recursive Descent Compiling

- Each major language structure has a corresponding recognizer routine.

Recursive Descent Compiling

- Each major language structure has a corresponding recognizer routine.
- These methods call each other as needed.

Recursive Descent Compiling

- Each major language structure has a corresponding recognizer routine.
- These methods call each other as needed.
- As the methods get called, they construct a parse tree.

Recursive Descent Compiling

- Each major language structure has a corresponding recognizer routine.
- These methods call each other as needed.
- As the methods get called, they construct a parse tree.
- Errors are detected as the recognizers execute.

Recursive Descent Compiling

- Each major language structure has a corresponding recognizer routine.
- These methods call each other as needed.
- As the methods get called, they construct a parse tree.
- Errors are detected as the recognizers execute.
- Limited in scope to $\mathrm{LL}(1)$ languages.

Outline

Language Properties

- ALGOL Inspired Language

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration
- Vectors for Lists of Variables

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration
- Vectors for Lists of Variables
- Structures

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration
- Vectors for Lists of Variables
- Structures
- Procedures

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration
- Vectors for Lists of Variables
- Structures
- Procedures
- Designed as a Teaching Language

Language Properties

- ALGOL Inspired Language
- Sequence Level Scoping
- Types are Inferred at Declaration
- Vectors for Lists of Variables
- Structures
- Procedures
- Designed as a Teaching Language
- Powerful Enough for Systems Programming

Variable Declarations

```
let x := 1                  !has type int i.e.integer
let y := 2.7                !has type real
let switch := x<pi          !has type bool i.e. boolean
let name := "Bill"          !has type string
let e=2.71828               !real constant
let lbl := "here"           !has type cstring
```

Structures

```
structre identifier(cstring name ;real val)
let var := identifier("x", 2.14)
```

Procedures

```
procedure count(cint s,e)
begin
  let x := s
  while x <= e do
  begin
    write x
    x := x + 1
  end
end

procedure convert(cint L,S,D->real)
    L+S/20+D/240
```

Outline

Closure

- **Closure** - A sequence of objects which close a set of objects.

Closure

- **Closure** - A sequence of objects which close a set of objects.
- We often speak of closure of grammars, languages, and sets.

Closure

- **Closure** - A sequence of objects which close a set of objects.
- We often speak of closure of grammars, languages, and sets.
- Computing closures reveal vital information about a language.

Closure

- **Closure** - A sequence of objects which close a set of objects.
- We often speak of closure of grammars, languages, and sets.
- Computing closures reveal vital information about a language.
- We will get more formal with closures later.

Alphabets and Languages

- An alphabet, A, is a finite set of symbols. For example:

$$A = \{a, b, c\}$$

Alphabets and Languages

- An alphabet, A, is a finite set of symbols. For example:

$$A = \{a, b, c\}$$

- A language, L, is the set of sequences or strings over some alphabet. For example, we may have:

$$L = A \times A$$

Alphabets and Languages

- An alphabet, A, is a finite set of symbols. For example:

$$A = \{a, b, c\}$$

- A language, L, is the set of sequences or strings over some alphabet. For example, we may have:

$$L = A \times A$$

- Expanding the above language yields:

$$L = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

Strings of Length k

- Often we wish to express arbitrarily long strings of alphabet A.

Strings of Length k

- Often we wish to express arbitrarily long strings of alphabet A.
- Strings of length k are written as A^k where k represents the number of times the Cartesian product is applied to A and itself.

Strings of Length k

- Often we wish to express arbitrarily long strings of alphabet A.
- Strings of length k are written as A^k where k represents the number of times the Cartesian product is applied to A and itself.
- For example:

$$A^3 = A \times A \times A$$

Represents all strings consisting of 3 symbols from A.

Strings of Length k

- Often we wish to express arbitrarily long strings of alphabet A.
- Strings of length k are written as A^k where k represents the number of times the Cartesian product is applied to A and itself.
- For example:

$$A^3 = A \times A \times A$$

  Represents all strings consisting of 3 symbols from A.
- A^0 is the empty string, we often give it the special symbol λ

Reflexive Transitive Closure

- Suppose we take A^k for all values $k = [0 \ldots \infty]$

Reflexive Transitive Closure

- Suppose we take A^k for all values $k = [0 \ldots \infty]$
- The union of every such set of sequences is called A^*

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

Reflexive Transitive Closure

- Suppose we take A^k for all values $k = [0 \ldots \infty]$
- The union of every such set of sequences is called A^*

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

- A^* is the reflexive transitive closure of A

Reflexive Transitive Closure

- Suppose we take A^k for all values $k = [0 \ldots \infty]$
- The union of every such set of sequences is called A^*

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

- A^* is the reflexive transitive closure of A
- Also known as the Kleene closure (after its definer, Stephen Kleene).

Reflexive Transitive Closure

- Suppose we take A^k for all values $k = [0 \ldots \infty]$
- The union of every such set of sequences is called A^*

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

- A^* is the reflexive transitive closure of A
- Also known as the Kleene closure (after its definer, Stephen Kleene).
- A^* is the language consisting of every possible string over the alphabet A.

Transitive Closure

- Sometimes, we wish to exclude λ from the set of strings. This is the A^+ closure.

Transitive Closure

- Sometimes, we wish to exclude λ from the set of strings. This is the A^+ closure.
- $A^+ = A^* - \lambda$

Transitive Closure

- Sometimes, we wish to exclude λ from the set of strings. This is the A^+ closure.
- $A^+ = A^* - \lambda$
- Fully stated:

$$A^+ = \bigcup_{n=1}^{\infty} A^n$$

Transitive Closure

- Sometimes, we wish to exclude λ from the set of strings. This is the A^+ closure.
- $A^+ = A^* - \lambda$
- Fully stated:

$$A^+ = \bigcup_{n=1}^{\infty} A^n$$

- This is called the Transitive Closure of A

Concatenation

- **Concatenation** is an operator which combines two strings.

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.
- λ is the unit element since $\forall s \in A^*$:

$$s.\lambda = \lambda.s = s$$

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.
- λ is the unit element since $\forall s \in A^*$:

$$s.\lambda = \lambda.s = s$$

- Concatenation is associative, but not commutative.

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.
- λ is the unit element since $\forall s \in A^*$:

$$s.\lambda = \lambda.s = s$$

- Concatenation is associative, but not commutative.
- **Closure** is the smallest set containing a given basic set closed under certain operations.

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.
- λ is the unit element since $\forall s \in A^*$:

$$s.\lambda = \lambda.s = s$$

- Concatenation is associative, but not commutative.
- **Closure** is the smallest set containing a given basic set closed under certain operations.
- Hence A^* is the reflexive transitive closure of A under the operation of concatenation.

Concatenation

- **Concatenation** is an operator which combines two strings.
- Concatenation (represented as a .) defines an algebra over A^*.
- λ is the unit element since $\forall s \in A^*$:

$$s.\lambda = \lambda.s = s$$

- Concatenation is associative, but not commutative.
- **Closure** is the smallest set containing a given basic set closed under certain operations.
- Hence A^* is the reflexive transitive closure of A under the operation of concatenation.
- Also, A^+ is the transitive closure of A under the operation of concatenation.

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.
- Usually, we think of the alphabet of a programming language as being its individual lexemes.

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.
- Usually, we think of the alphabet of a programming language as being its individual lexemes.
- For almost all programming languages:

$$L \subset A^*$$

Where A is the alphabet of lexemes of the language.

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.
- Usually, we think of the alphabet of a programming language as being its individual lexemes.
- For almost all programming languages:

$$L \subset A^*$$

  Where A is the alphabet of lexemes of the language.
- A compiler, is therefore formally stated as a device which, given a string s, a source language L, and an object language L':
  $\forall s \in A^*$:

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.
- Usually, we think of the alphabet of a programming language as being its individual lexemes.
- For almost all programming languages:

$$L \subset A^*$$

  Where A is the alphabet of lexemes of the language.
- A compiler, is therefore formally stated as a device which, given a string s, a source language L, and an object language L':
  $\forall s \in A^*$:
  1. Decides $s \stackrel{?}{\in} L$

Languages and Strings

- If we think of a programming language, L is the set of all strings which form a valid program.
- Usually, we think of the alphabet of a programming language as being its individual lexemes.
- For almost all programming languages:

$$L \subset A^*$$

  Where A is the alphabet of lexemes of the language.
- A compiler, is therefore formally stated as a device which, given a string s, a source language L, and an object language L':
  $\forall s \in A^*$:
  1. Decides $s \stackrel{?}{\in} L$
  2. Computes the function $L \mapsto L'$ on s