ALGONQUIN COLLEGE

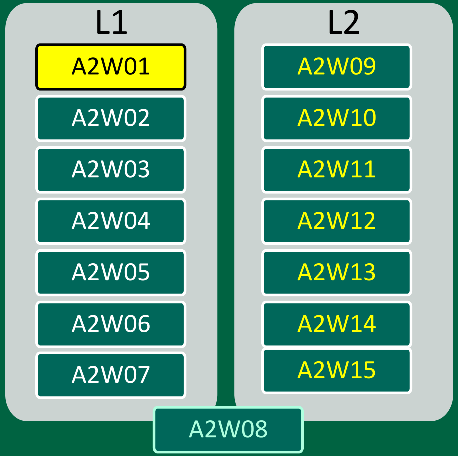
CST8152

Lecture 01

Compilers – General Concepts

Week 1: Introduction

- *What is a Compiler?*
- *What about Programming Languages?*
- *Past, Present and Future.*

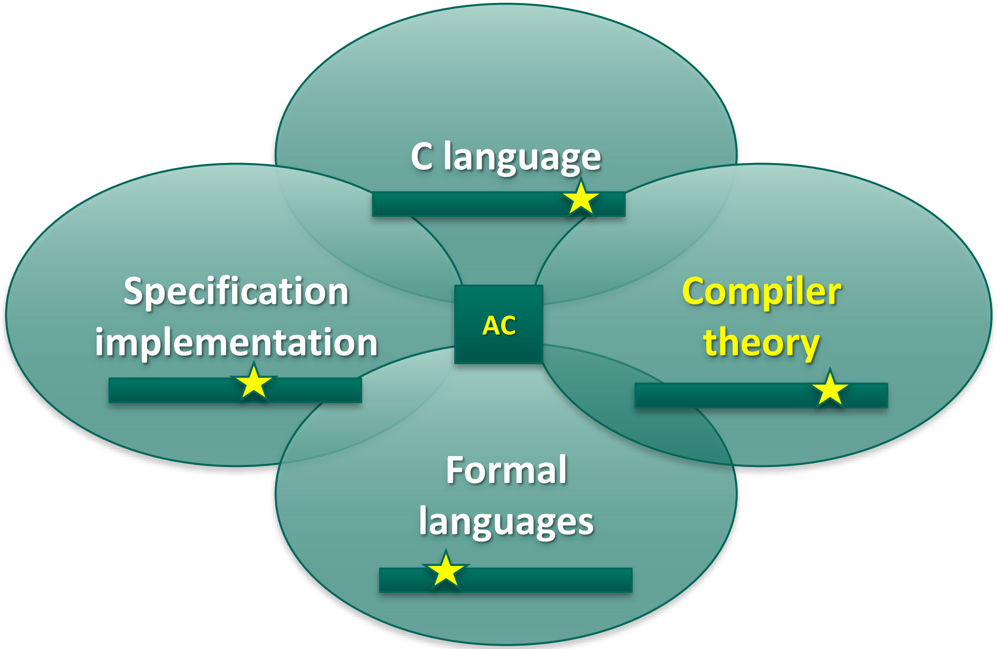


ALGONQUIN COLLEGE

Compilers – Week 1

What is a Compiler?

Lets start…


C language
Specification implementation
AC
Compiler theory
Formal languages

1.1. Initial Concepts

- A **Compiler** is a program that runs on some computer architecture under some operating system and transforms (translates) an input program (source program) written in some programming language into an output program (target program) expressed in different programming language.
- A **Programming Language** is a notational system for describing computations in machine-readable and human-readable form.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.1. Initial Concepts

- **Computation** in general is any process that can be carried by a computer. Programming Languages must provide two types of abstractions:
  - **Data abstractions** and
  - **Control abstractions**.
- **Note**:
  - Niklaus Wirth (Pascal language creator) defined a Programming language as: **Data structures + Algorithms.**
  - Decades after, **OO** defined entities as composed by **properties** and **methods**.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.2. Importance of Compilers

- **Compilers are used by all programmers**. "*A good craftsman should know his tools.*"
- **Compilers elements and techniques are used in almost every application**. See some domain-specific language (DSL).
  - Writing a parser for XML, HTML, or some other structured data file is a common task.
  - Scanning and parsing a command or user input line is a very common task.
  - Looking for a specific word or sentence in a text is a very common task.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.2. Importance of Compilers

- **Compilers are an excellent "capstone" or "focal" programming project.**
  - Writing a compiler requires an understanding of almost all of the basic computer science subfields.
- **Computer Science** specific topic.
  - It can demonstrate your proficiency in Computer Science.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.3. Brief History

- **1830 – 40:** Analytical Engine (Charles Babbage). His wife Ada Augusta, wrote the first programs.
- **1950:** FORTRAN, COBOL, Algol60, LISP.
- **1960:** PL/1, SNOBOL, Simula, BASIC.
- **1970:** Pascal (1971), C (1972).
- **1980:** Ada, Modula, Smalltalk-80, C++, Objective C, Object Pascal, Eiffel, Oberon, Scheme.
- **1990:** Java, Haskell, Javascript, PHP, Perl, Python, Ruby, Lua…
- **2000:** C#, Scala , F#, Groovy, Go, D, R, Clojure, Swift, Kotlin …
- **2010:** Rust, Julia, …
- **2020:** GPT-3 (OpenAI) …

Source: Wanner Bros.

ALGONQUIN COLLEGE

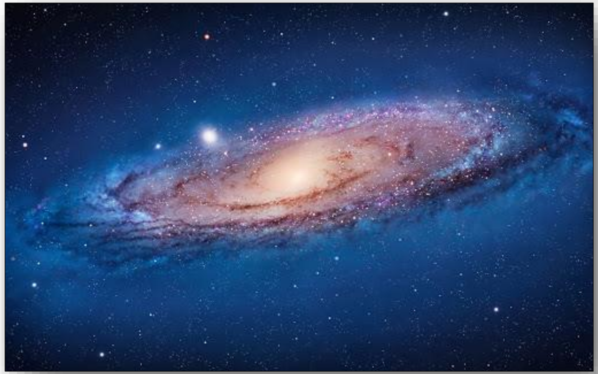
1.3. Brief History

- **To think about the future:**
  - ➢ Facebook AI Creates Its Own Language In Creepy Preview Of Our Potential Future:
    - https://www.forbes.com/sites/tonybradley/2017/07/31/facebook-ai-creates-its-own-language-in-creepy-preview-of-our-potential-future/
  - ➢ The truth behind Facebook AI inventing a new language:
    - https://towardsdatascience.com/the-truth-behind-facebook-ai-inventing-a-new-language-37c5d680e5a7
  - ➢ OpenAI API:
    - https://openai.com/blog/openai-api/
  - ➢ GPT-3 Demo:
    - https://www.youtube.com/watch?v=8psgEDhT1MM
  - ➢ GPT-3 Paper:
    - https://arxiv.org/pdf/2005.14165.pdf
  - ➢ Kevin Lacker tests:
    - https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html

Source: https://encrypted-tbn0.gstatic.com/images?q=tbn%3AANd9GcRo3UP44TbbvFrlrRn43YRi3KwwK2yox3vmig&usqp=CAU

1.4. Computational Paradigms

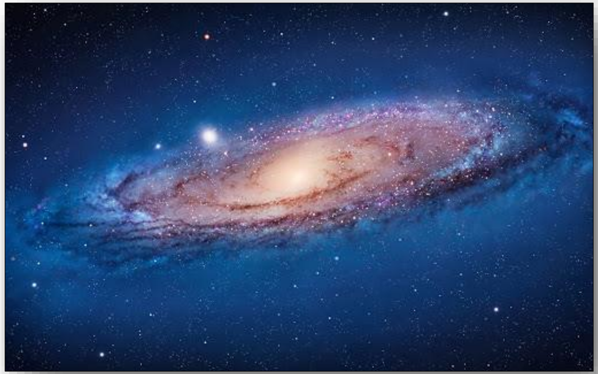
- **Imperative or Procedural Programming:**
  - FORTRAN, COBOL, ALGOL, BASIC, PL, C, ADA, Modula.
- **Functional Programming:**
  - LISP, Scheme, ML (Meta Language), Miranda, Haskell, F#, Clojure.
- **Logic Programming:**
  - Prolog
- **Object-Oriented Programming:**
  - SIMULA, Smaltalk, C++, Objective C, Eiffel, JAVA, C#.
- **Scripting Languages:**
  - Perl, Python, Tcl/Tk, Javascript, Rexx, Visual Basic, PHP



Source: NASA

1.4. Special Purpose Languages

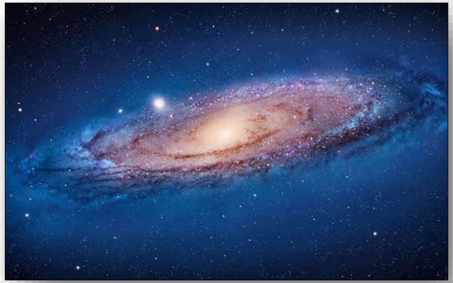
- **Database Query Languages:**
  - SQL
- **Simulation Languages:**
  - Simula, GPSS, SIMSCRIPT
- **Silicon Design Languages:**
  - VRML, VHDL,SystemC (C++), SpecC(C)
- **Graphics Design Languages**
  - GRAF
- **Real-time Languages:**
  - RT-FORTRAN, BCL, Embedded-C, Embedded Java



Source: NASA

1.4. DSL (Domain Specific Languages)

- **Definition**
  - *A domain-specific language* (DSL) is a computer language specialized to a particular application domain. This is in contrast to a *general-purpose language* (GPL), which is broadly applicable across domains.
- **Design Goals:**
  - Domain-specific languages are *less comprehensive*.
  - Domain-specific languages are much *more expressive* in their domain.
  - Domain-specific languages should exhibit *minimal redundancy*.
- **Examples:**
  - OS Shells, Wiki environments, OpenGL, Markup Languages…



1.5. Some Definitions

- **Compiler Input:**
  - *Source program*
  - Configuration parameters or pragmatics (#pragma directives)
- **Compiler Output:**
  - *Target program*
  - Error messages
  - Information accompanying the target program – external symbol tables, cross- reference tables.
- **Target Program:**
  - Low-Level Code (Language)

Source: https://toCwardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.5. Some Definitions

- **Target Low-Level Code Type:**
  - Pure Machine Code
  - Augmented Machine Code
  - Virtual Machine Code
- **Target Low-Level Code Format:**
  - Assembly or Pseudo-assembly Language Format,
  - Relocatable Binary Format
  - Memory-Image Format (Load & Go)
- **Run-time Environment:**
  - *Fully Static* Environment
  - *Fully Dynamic* Environment
  - *Mixed Environment* – Stacked-based **environment**

Compiler Related Applications

- Editors, Word Processors, Command Interpreters, Formatting Printers, XML Parser, and almost all applications – big and small.

Source: https://toCwardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

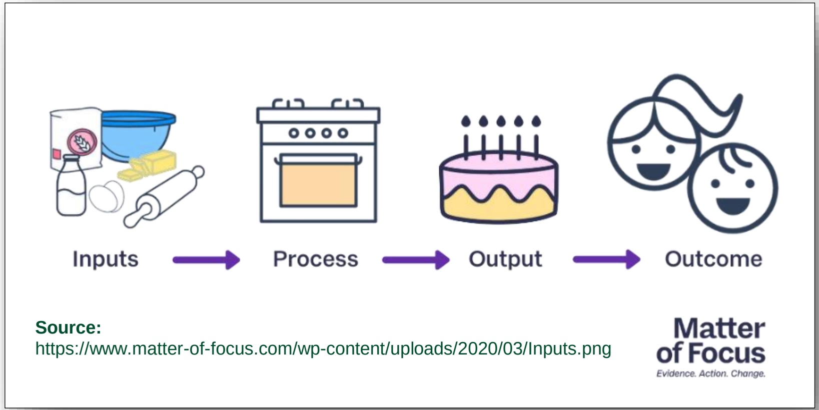
ALGONQUIN COLLEGE

Compilers – Week 1

Concluding

Review

- *Compiler definition*
- *Basic overview (historical and trends)*
- *Computational paradigms*
- *Elements of compiler*



Source: https://www.matter-of-focus.com/wp-content/uploads/2020/03/Inputs.png

Source: https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Some Questions

1. *What is the importance of compilers?*
2. *Summarize the functionality of a compiler.*
3. *Identify some challenges to create a compiler.*

Source: https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Open questions…

- Any doubts / questions?
- How we are until now?

Image URL: https://cdn-az.allevents.in/events3/banners/26b150363d5757da8578fa6a1481585a368f12d4247adfe95837e6ea6c5ab2af-rimg-w1200-h549-gmir.jpg?v=1569691155

Compilers – Week 1

Thank you for your attention!

Contact: sousap@algonquincollege.com

ALGONQUIN COLLEGE

ALGONQUIN COLLEGE

CST8152

Lecture 02

Context of Compiler

Week 2: Context of a Compiler

- *Overview - Context*

L1	L2
A2W01	A2W09
A2W02	A2W10
A2W03	A2W11
A2W04	A2W12
A2W05	A2W13
A2W06	A2W14
A2W07	A2W15

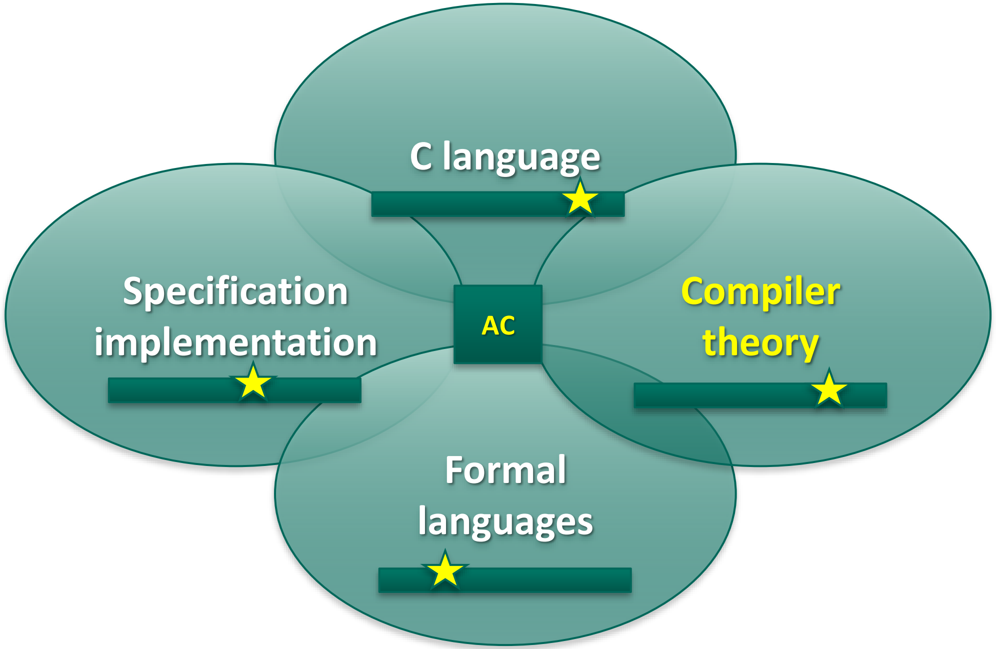
A2W08

ALGONQUIN COLLEGE

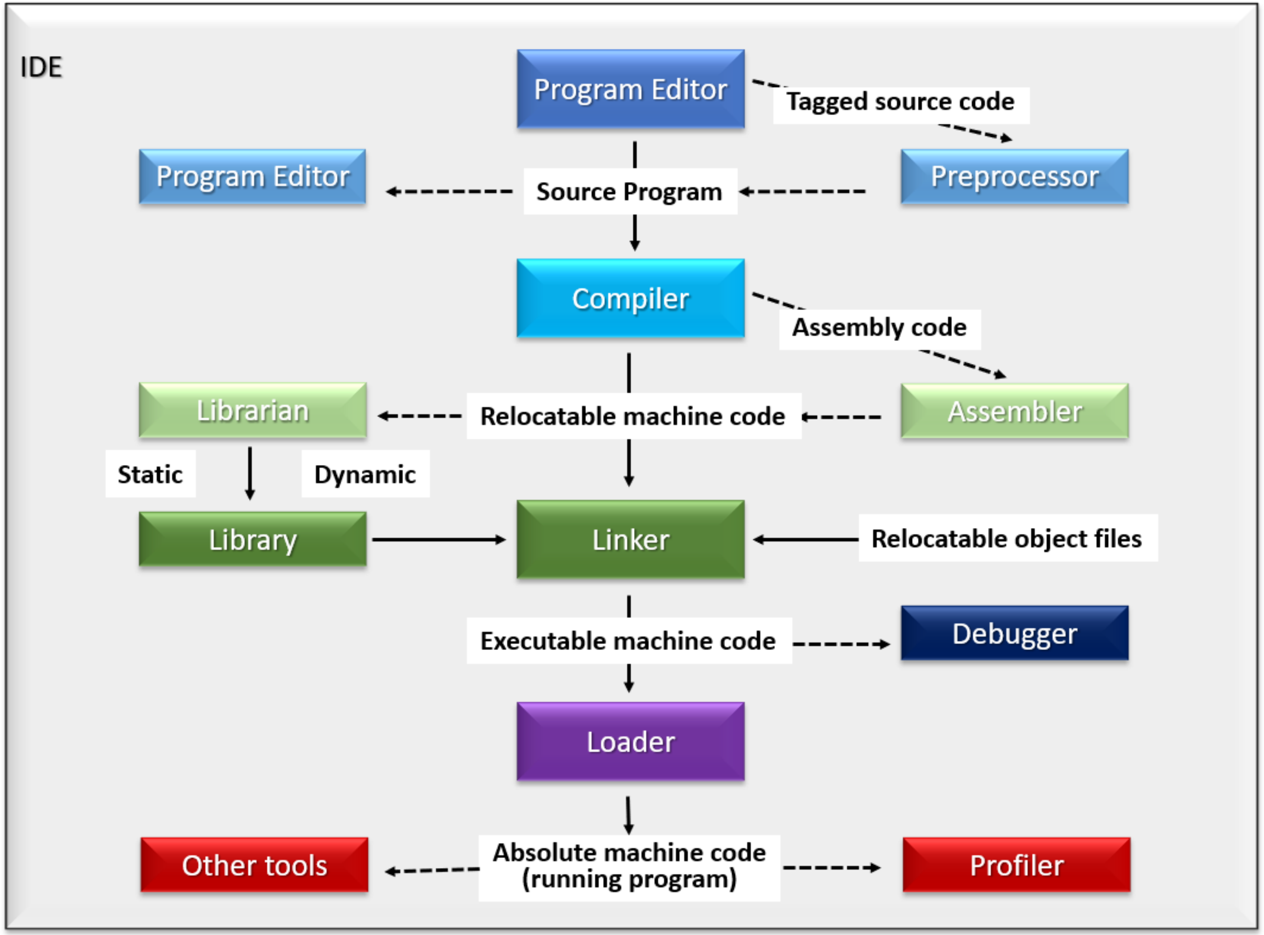
Compilers – Week 2

Context of a Compiler

Lets start…


C language
Specification implementation
AC
Compiler theory
Formal languages

2.1. General Diagram



Source: Algonquin College (Svillen Ranev)

2.1. Definitions

- **<u>Assembler</u>**: Assemblers are simple compilers which translate assembly language into machine code.
- **<u>Compiler</u>**: Translate the text of the program in another language - usually assembler or some form of machine code.
- **<u>Debugger</u>**: Allows the user to trace the execution of a program statement by statement and inspect the content of different parts of the program memory.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

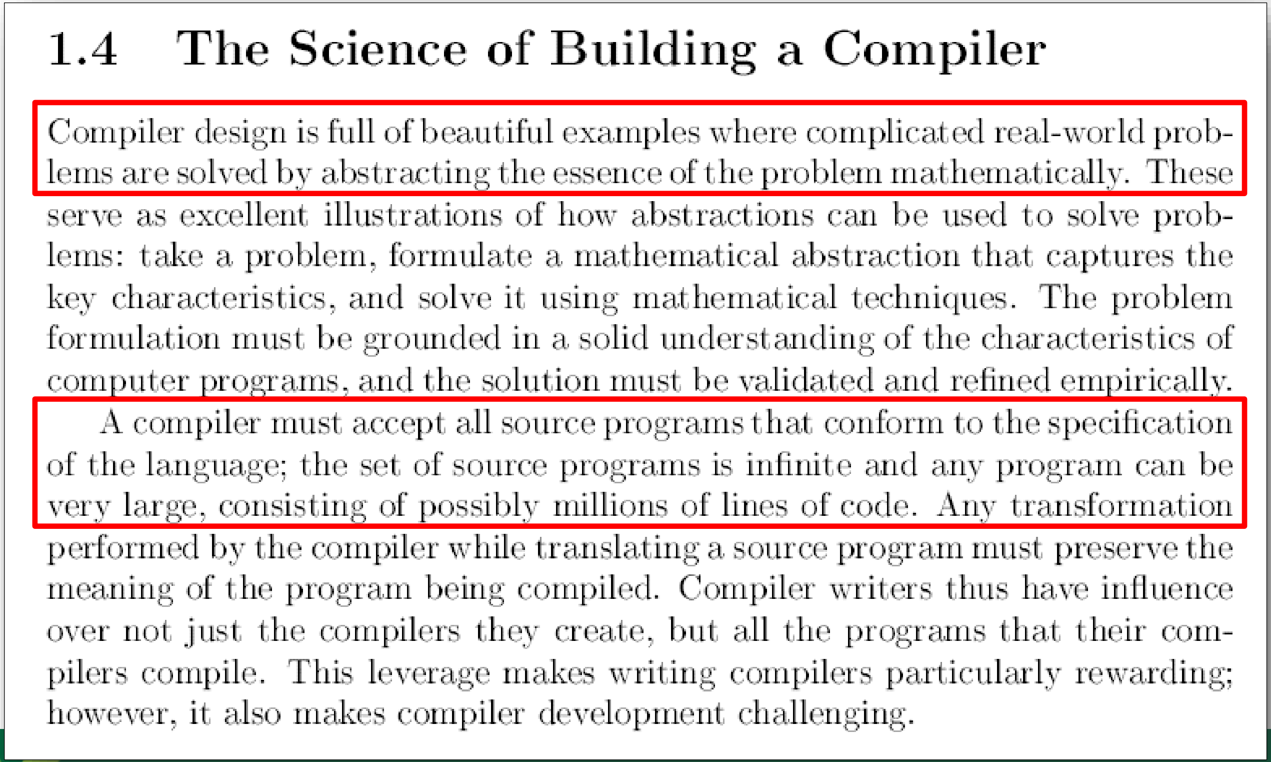
2.1. Definitions

- **Librarian**: Allows creating and maintaining libraries of pre-compiled component which can be used later without the need to be compiled again.
- **Linker**: Combines (links) all necessary components of a program into some executable form. Not all programming languages require linkers.
- **Loader**: Loads an executable program and passes the control to the program.

2.2. Other Tools

- Automatic or Unit testers (JUnit),
- Code Inspectors and Analyzers,
- Error loggers,
- Make and build scripting tools (Ant),
- Profilers,
- Project Managers (Maven)
- Refactoring tools,
- Run-time Inspectors,
- Style Formatters,
- Task Managers (Mylyn),
- Version Control and, Collaboration (Git).

2.3. General View



1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

2.3. Example Hybrid Compiler

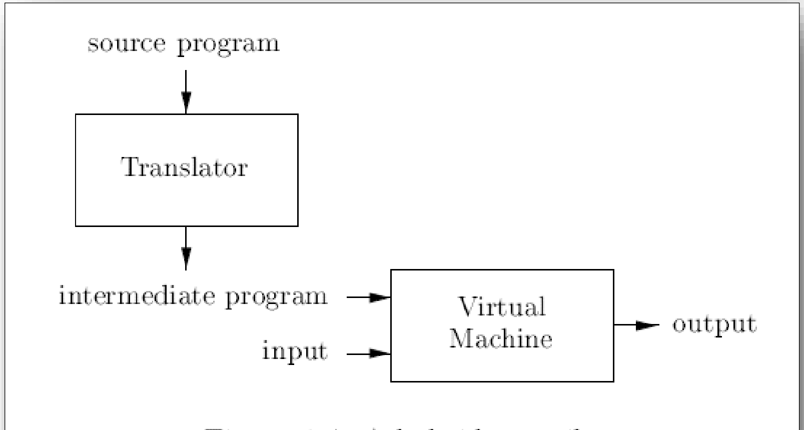


Figure 1.4: A hybrid compiler

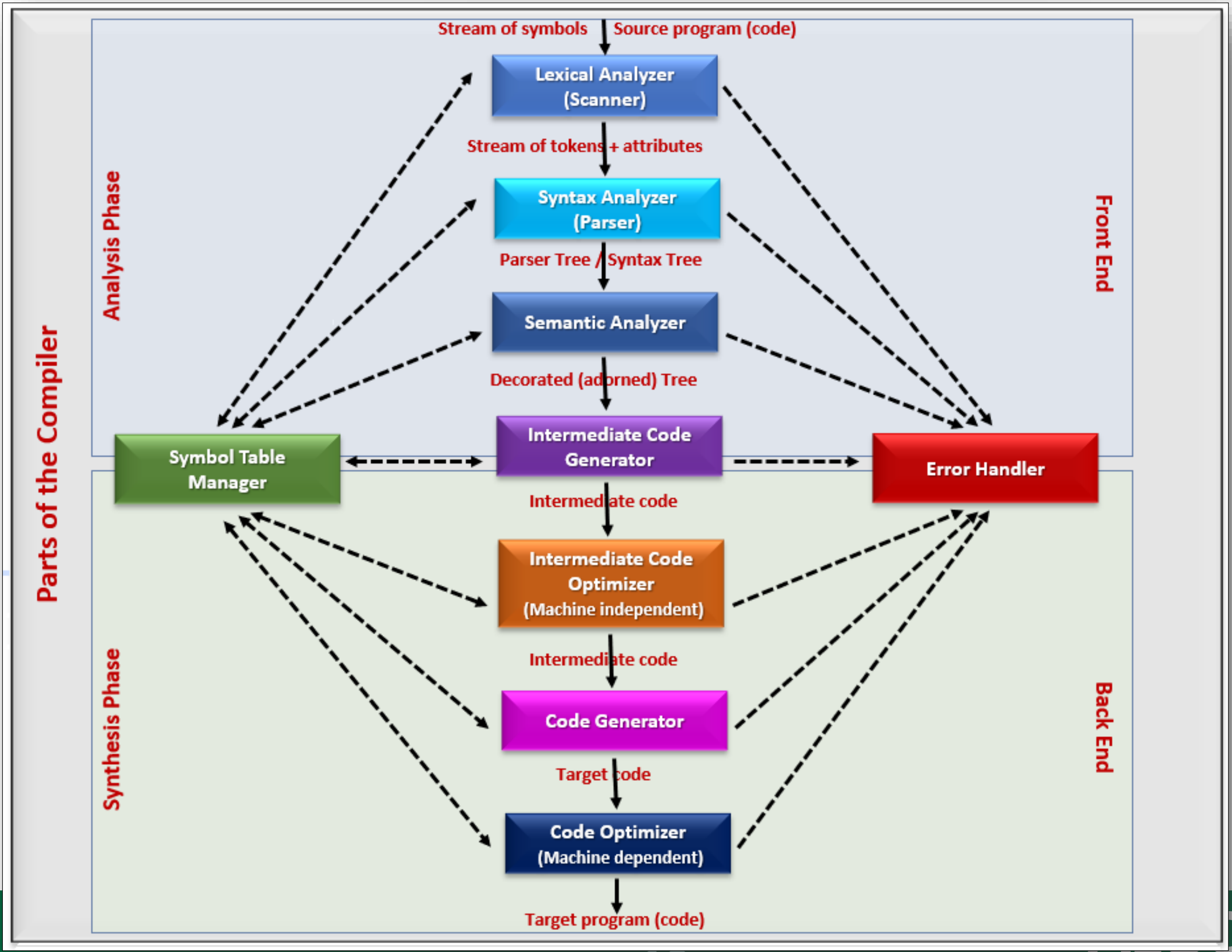
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("This will be printed");
    }
}
```

```
> javap HelloWorld.class
Compiled from "HelloWorld.java"
public class HelloWorld {
  public HelloWorld();
  public static void main(java.lang.String[]);
}
```

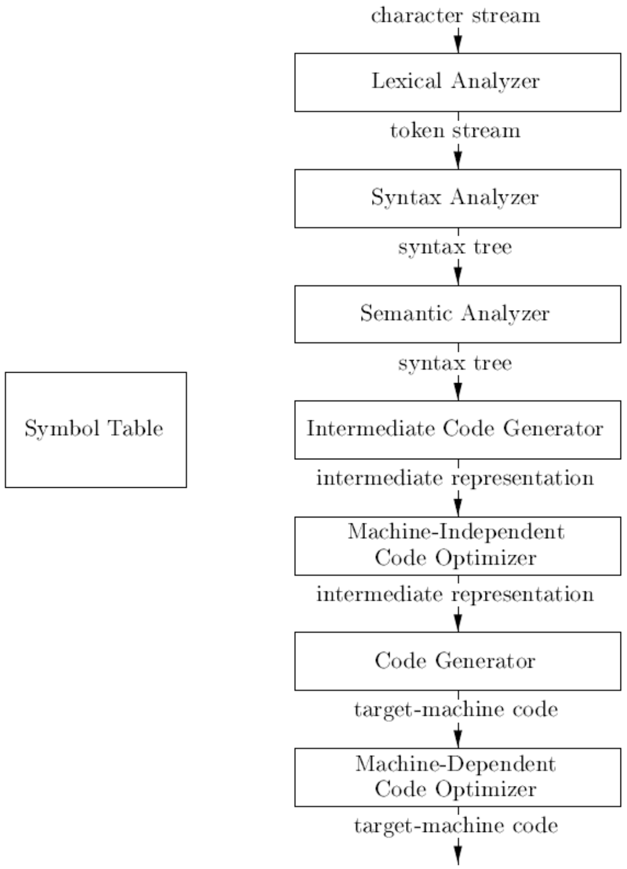
2.3. Example Hybrid Compiler

```
> javap -c HelloWorld.class
Compiled from "HelloWorld.java"
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String This will be printed
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```


Parts of the Compiler
Analysis Phase
Synthesis Phase
Front End
Back End
Stream of symbols
Source program (code)
Lexical Analyzer (Scanner)
Stream of tokens + attributes
Syntax Analyzer (Parser)
Parser Tree / Syntax Tree
Semantic Analyzer
Decorated (adorned) Tree
Intermediate Code Generator
Symbol Table Manager
Error Handler
Intermediate code
Intermediate Code Optimizer (Machine independent)
Intermediate code
Code Generator
Target code
Code Optimizer (Machine dependent)
Target program (code)

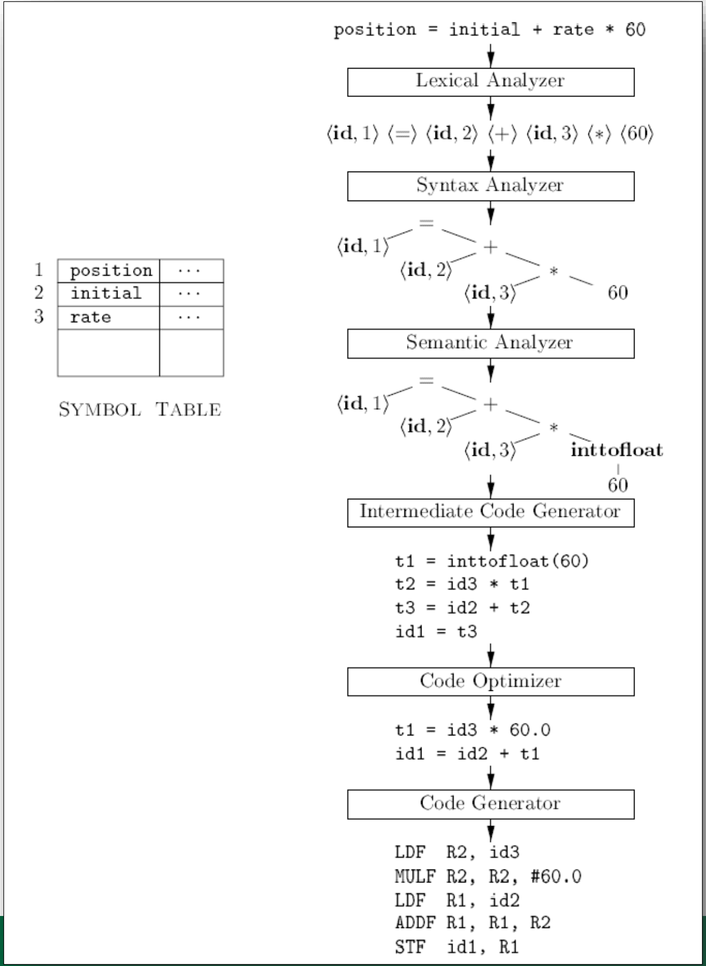
2.3. General View



$$\begin{tikzcd}[row sep=small]
\text{Symbol Table} & \text{character stream} \arrow[d] \\
& \boxed{\text{Lexical Analyzer}} \arrow[d, "\text{token stream}"] \\
& \boxed{\text{Syntax Analyzer}} \arrow[d, "\text{syntax tree}"] \\
& \boxed{\text{Semantic Analyzer}} \arrow[d, "\text{syntax tree}"] \\
& \boxed{\text{Intermediate Code Generator}} \arrow[d, "\text{intermediate representation}"] \\
& \boxed{\text{Machine-Independent Code Optimizer}} \arrow[d, "\text{intermediate representation}"] \\
& \boxed{\text{Code Generator}} \arrow[d, "\text{target-machine code}"] \\
& \boxed{\text{Machine-Dependent Code Optimizer}} \arrow[d, "\text{target-machine code}"] \\
& {}
\end{tikzcd}$$

Figure 1.6: Phases of a compiler

2.3. General View



```
position = initial + rate * 60
```

1	position	···
2	initial	···
3	rate	···

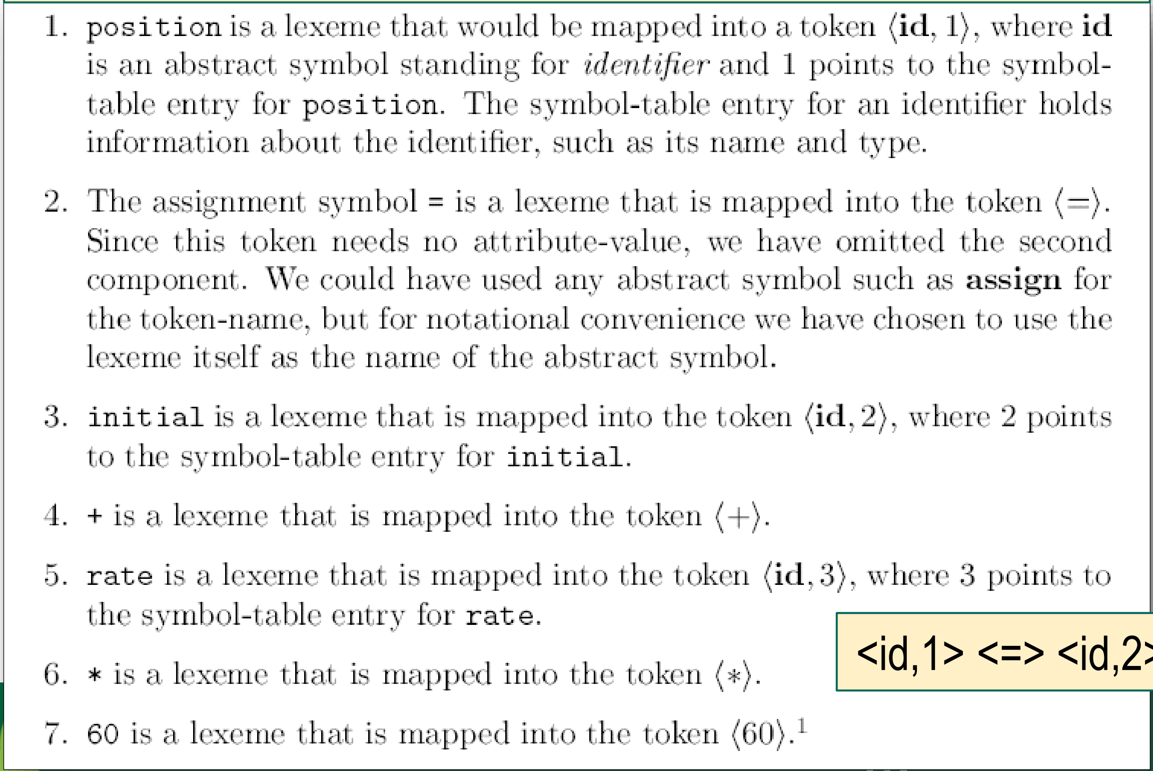
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

2.3. General View

position = initial + rate * 60



1. position is a lexeme that would be mapped into a token ⟨**id**, 1⟩, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token ⟨=⟩. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. initial is a lexeme that is mapped into the token ⟨**id**, 2⟩, where 2 points to the symbol-table entry for initial.

4. + is a lexeme that is mapped into the token ⟨+⟩.

5. rate is a lexeme that is mapped into the token ⟨**id**, 3⟩, where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token ⟨∗⟩.

7. 60 is a lexeme that is mapped into the token ⟨60⟩.[1]

<id,1> <=> <id,2> <+> <id,3> <*> <60>

Compilers – Week 1

Concluding

Review

- *Define the elements of the context of the diagram.*

Some Questions

1. *Describe the compilation process using the previous concepts.*
2. *Give some errors that can happen in this process.*
3. *Propose a strategy to identify and fix (when possible) these errors.*

Source: https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Open questions…

- Any doubts / questions?
- How we are until now?

Image URL: https://cdn-az.allevents.in/events3/banners/26b150363d5757da8578fa6a1481585a368f12d4247adfe95837e6ea6c5ab2af-rimg-w1200-h549-gmir.jpg?v=1569691155

Compilers – Week 2

Thank you for your attention!

Contact: sousap@algonquincollege.com

ALGONQUIN COLLEGE

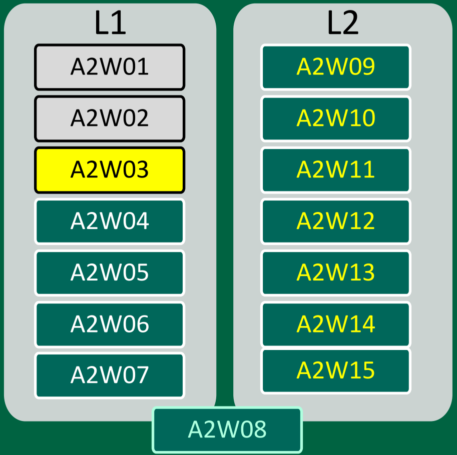
CST8152

Lecture 03

C Language and Planning Compiler Development

Week 3: C Language Context

- *Some "pieces" of C-cake…*

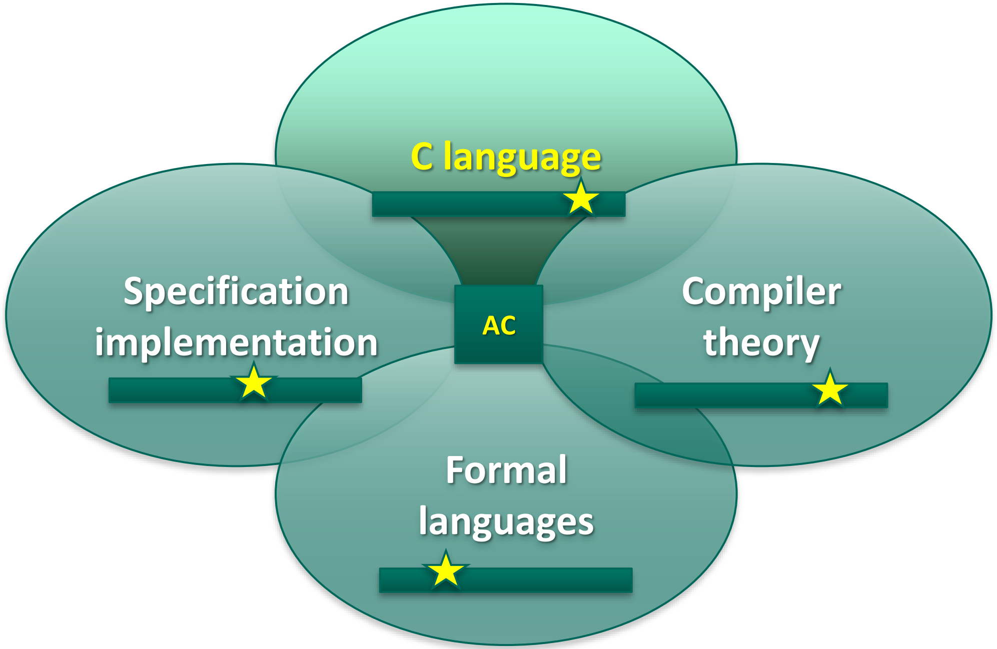


ALGONQUIN COLLEGE

Compilers – Week 3

Pre-Processor

Let's start…

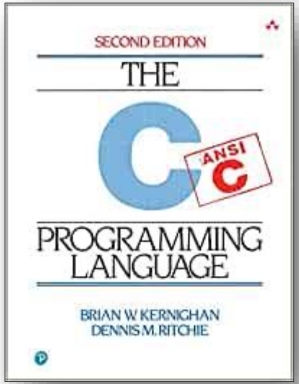

C language
Specification implementation
AC
Compiler theory
Formal languages

3.1. Basic Notation

What is pre-processing?

- C provides certain language facilities by means of a **preprocessor**, which is conceptionally a separate first step in compilation.
  - The two most frequently used features are #include, to include the contents of a file during compilation, and #define, to replace a token by an arbitrary sequence of characters.
  - Other features include conditional compilation and macros with arguments.

Note: All these facilities will be used in A1.



3.1. Basic Notation

Samples

- **File Inclusion:**

```
#include <filename> /* standard libraries header files */
#include "filename" /* user defined header files */
```

- **Macro definitions:**

```
#define NAME /* defines name */
#undef NAME /* undefines name */
#define NAME Replacement /* replaces NAME with replacement */
#define name(list) expression /* inline functions or macro */
```

Remember: It is a good practice to create constants in your header file.

C
ANSI

3.1. Basic Notation

Samples:

- **Conditional Processing:**

```
#if constant-expression
#ifdef NAME
#if defined(NAME)
#ifndef NAME
#if !defined(NAME)
#elif constant-expression /* optional */
#else /* optional */
#endif /* must be present */
```

C

ANSI

ALGONQUIN COLLEGE

3.1. Basic Notation

Samples:

- **Pragma Definition:**

```
#pragma directive /* sets different compiler options */
```

The '#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The forms of this directive (commonly known as *pragmas*) specified by C standard are prefixed with STDC.

https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html

C

ANSI

3.1. Basic Notation

Other directives:

- **Other:**

```
# /* null directive – no any action or “” enclosure */
## /* token passing – concatenation */
#line constant /* changes the line number of the source */
#error text /* compiler displays error message including text */
```

- **Predefined macros in ANSI C:**

```
__LINE__ , __FILE__ , __TIME__ , __DATE__ , __STD__
```

C
ANSI

ALGONQUIN COLLEGE

3.2. Examples

Look what you can do in A1:

```
#include <limits.h>
```

```
#include “buffer.h”
```

```
#ifndef BUFFER_H_
#define BUFFER_H_
... declaration
#endif
```

```
#undef getchar
int getchar(void){…}
```

C

ANSI

3.3. Debugging

Using Preprocessing to debug…

```
#define DEBUG
```

```
#ifdef DEBUG
  printf(“The size is negative: %d”, size)
#endif
```

```
#undef DEBUG
```

C

ANSI

3.4. Constants / Functions

Some useful samples

```
#define FAIL -1
```

- **Warning**: If FAIL is enclosed in double quotes it will not be replaced. If, however, a parameter name is preceded by a # symbol in the replacement text, the combination will be replaced by the actual argument enclosed in quotes.

```
#define reprint(fexp) printf(#fexp " = %9.3f\n",fexp)
```

```
reprint(x+y);   ➡   printf("x+y" " = %9.3f\n",x+y);
```

C ANSI

ALGONQUIN COLLEGE

3.5. Macros

Macros as Functions

- Ex1:

```
#define WCALC(y,x) ((y + x) / (x -y) * (INT_MAX - 77x))
```

- Ex2:

```
#define sum(x,y) x+y
```
$\Rightarrow$
```
a = a + sum(b,c);
```

- Caution:

```
a = a*sum(b+c,d+e)/m;
```
$\Rightarrow$
```
a = a * b+c + d+e / m;
```

- To solve:
```
#define sum(x,y) ((x) + (y))
```

C

ANSI

3.5. Macros

More samples

- To create variable names:

```
#define STR(a,b) a ## b
...
STR(str,1) = strcat(STR(str,2),
STR(my,name));
```

⬇

```
str1 = strcat(str2, myname);
```

C
ANSI

3.6. Including appropriate header

More samples

- To create variable names:

```
#define PLATFORM 10
...
#if PLATFORM == __MSDOS__ * 10
#define HEADER "dos.h"
#elif PLATFORM == __WIN32__ * 9
#define HEADER "win.h"
#else
#define HEADER "sys.h"
#endif
#include HEADER
```

C

ANSI

3.7. Enforcing data

More samples

```
#define BORLAND_16
#ifdef BORLAND_16
typedef int int_2b;
typedef long long_4b;
#endif
#ifdef MSVS_16
typedef short int_2b;
typedef int long_4b;
#endif
```

```
printf("Int size: %d Long size:
   %d\n",sizeof(int_2b),sizeof(long_4b));
```

C
ANSI

3.8. Pragmas / Pragmatics

More samples

```
#pragma inline
```

Notes

- If a compiler does not recognize the `pragma` directive, it simply ignores it.
- This convention allows different compilers to use different `pragma` directives

C

ANSI

3.9. Bit-wise Operations

General View

- A bitwise operation operates on one or more bit patterns or binary numerals at the level of their individual bits.
    - It is a fast and simple action, directly supported by the processor, and is used to manipulate values for comparisons and calculations.
- **Note**: Bitwise operators treat their operands as a sequence of bits (zeroes and ones), rather than as decimal numbers. They operate on a single bit in the same location of the operands.

C
ANSI

3.9. Bit-wise Operations

Samples

- Single values:
  - c = 1:

0	0	0	0	0	0	0	0	1

  - c = -1:

1	1	1	1	1	1	1	1	1

- **Truth table – Boolean operations**

X1	X2	&	|	^	~X1	~X2
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

C
ANSI

3.9. Bit-wise Operations

Samples

Setting flag to 0

Flag variable

1	1	1	1	1	1	1	1	1

Mask:

0	0	0	0	0	0	0	0	0

AND operation = Result

0	0	0	0	0	0	0	0	0

C

ANSI

3.9. Bit-wise Operations

C ANSI

Samples

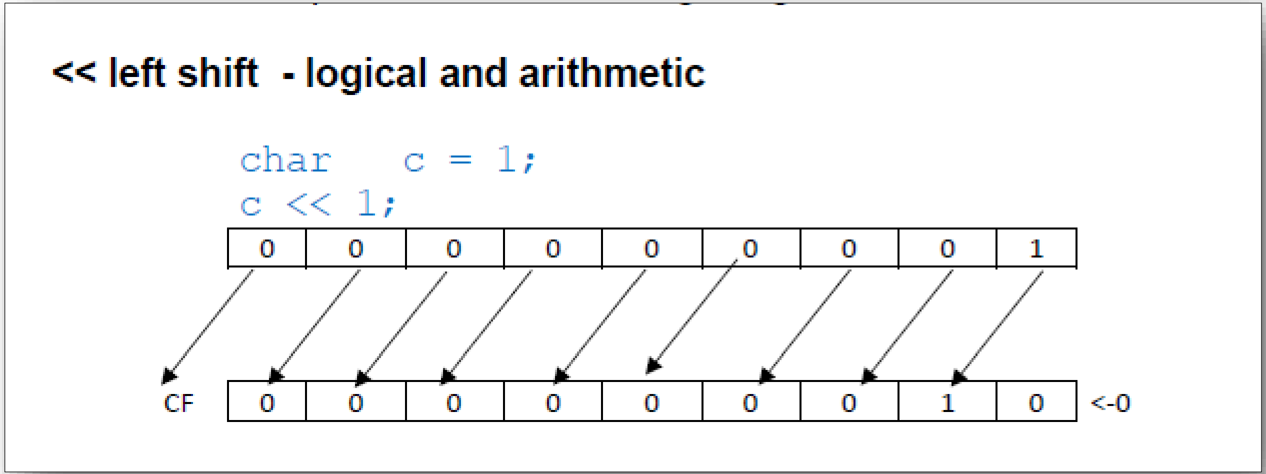
Setting flag to 1									
Flag variable	1	1	1	1	1	1	1	1	1
Mask:	0	0	0	0	0	0	0	0	0
AND operation = Result	0	0	0	0	0	0	0	0	0
Mask:	0	0	0	0	0	0	0	0	1
OR operation = Result	0	0	0	0	0	0	0	0	1
Checking flag to 1									
Flag variable	1	1	1	1	1	1	1	1	1
Mask:	0	0	0	0	0	0	0	0	1
AND operation = Result	0	0	0	0	0	0	0	0	1

3.9. Bit-wise Operations

C

ANSI

Samples


<< left shift - logical and arithmetic

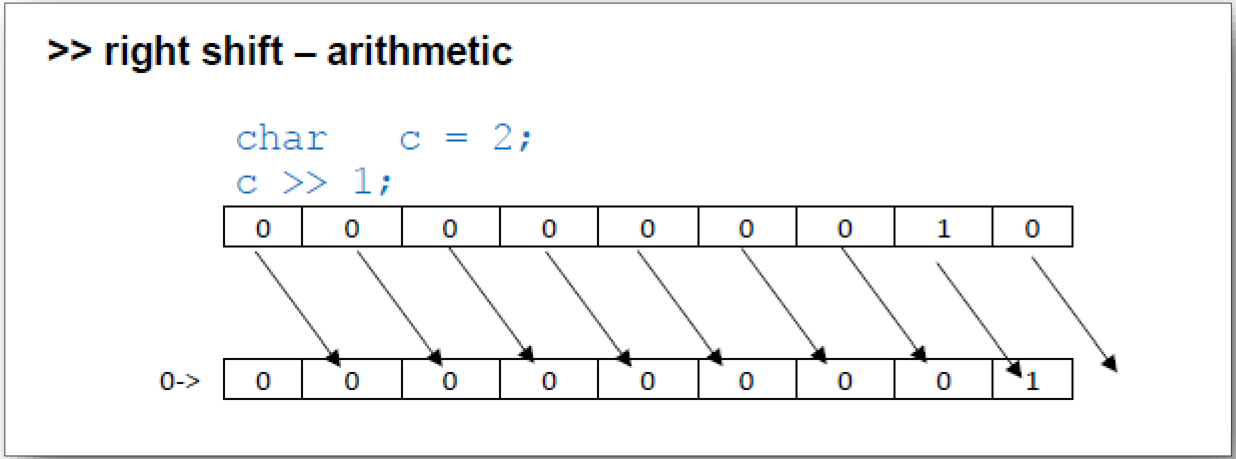
```
char   c = 1;
c << 1;
```

ALGONQUIN COLLEGE

3.9. Bit-wise Operations

C ANSI

Samples

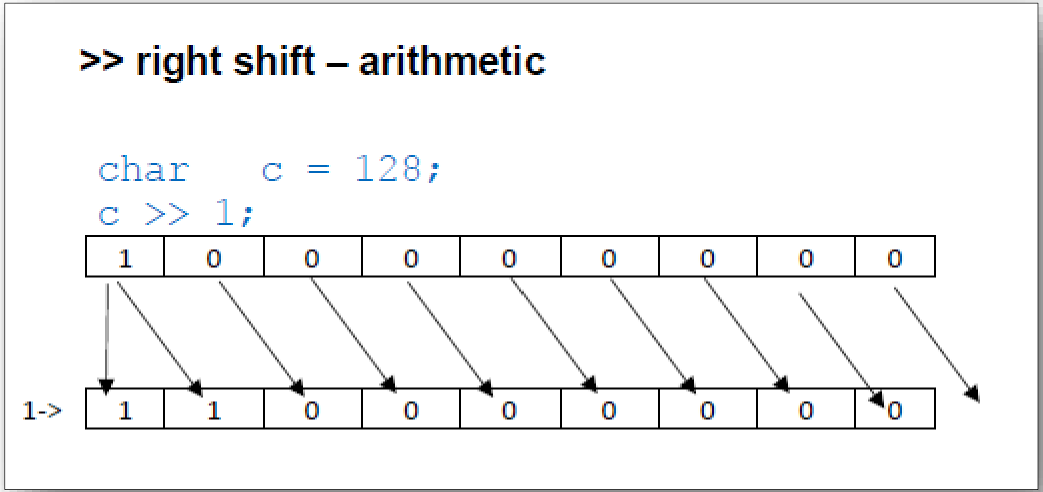


```
char   c = 2;
c >> 1;
```

3.9. Bit-wise Operations

ANSI

Samples


>> right shift – arithmetic

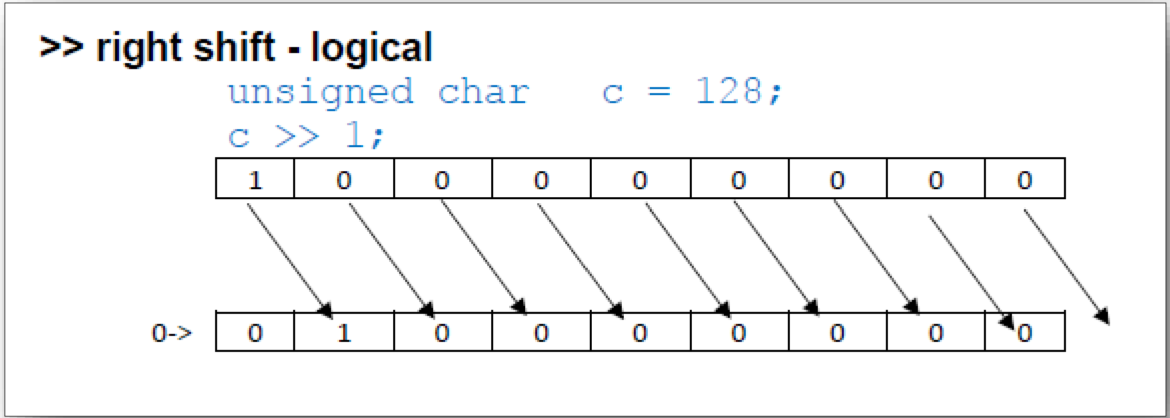
```
char    c = 128;
c >> 1;
```

3.9. Bit-wise Operations

C

ANSI

Samples


>> right shift - logical

```
unsigned char   c = 128;
c >> 1;
```

3.10. Bit Fields

C

ANSI

Samples

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:2;
} pack;
```

Note: Check the buffer structure for A1!

3.11. User defined Datatypes

C

ANSI

Samples

```
struct S {int a; char c};
struct {int a; char c} s;
```

```
struct S s;
```

- **NOTE:** To avoid using struct keyword all the time, use typedef.

```
typedef struct Structure {int a; char c} S;
S s;
```

Compilers – Week 3

C Lang Var Attributes

C-Language Att Var (1)

- **Definition:**
- External objects: variables or functions
  - Variables:
    - "External": Defined outside any function;
    - "Internal": Local var.
  - Functions:
    - Always external;

Think about this [3]:

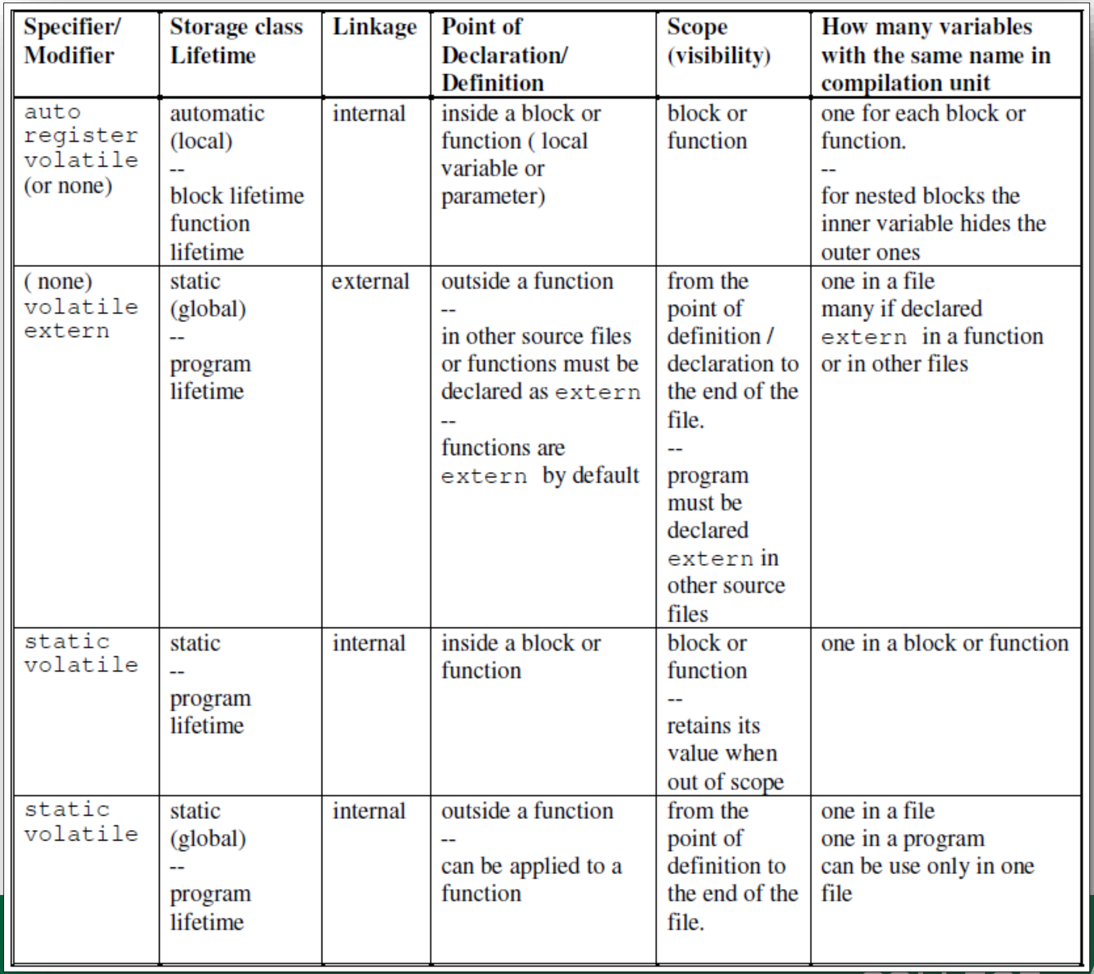
- Multiple source file program are compiled one by one.

Variable attributes:

C variables have the following attributes:

- Identifier (name): Obey naming rules;
- Type: See typedef, modifiers and specifiers.
- Size: See data range;
- Mutability: How initial values are changed;
  - Keywords: const, register, volatile, static:
- Storage class (lifetime):
  - Keywords: auto, register, extern, static:
- Scope (visibility):
  - Scope types: file, function (labels), block and declaration (function arguments)
- Linkage: Multiple source program;
  - Keywords: static, extern.

C-Language Att Var (2)



Specifier/ Modifier	Storage class Lifetime	Linkage	Point of Declaration/ Definition	Scope (visibility)	How many variables with the same name in compilation unit
`auto` `register` `volatile` (or none)	automatic (local) -- block lifetime function lifetime	internal	inside a block or function (local variable or parameter)	block or function	one for each block or function. -- for nested blocks the inner variable hides the outer ones
(none) `volatile` `extern`	static (global) -- program lifetime	external	outside a function -- in other source files or functions must be declared as `extern` -- functions are `extern` by default	from the point of definition / declaration to the end of the file. -- program must be declared `extern` in other source files	one in a file many if declared `extern` in a function or in other files
`static` `volatile`	static -- program lifetime	internal	inside a block or function	block or function -- retains its value when out of scope	one in a block or function
`static` `volatile`	static (global) -- program lifetime	internal	outside a function -- can be applied to a function	from the point of definition to the end of the file.	one in a file one in a program can be use only in one file

Compilers – Week 3

Concluding

ALGONQUIN COLLEGE

Review

- *Define the elements of the context of the diagram.*

Source:
https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Some Questions

1. *Identify what you are going to use in Assignment 1 (A1).*
2. *Exemplify bitwise operations in A1.*
3. *Give an idea about macro required to A1.*

Open questions…

- Any doubts / questions?
- How are we until now?

Image URL: https://cdn-az.allevents.in/events3/banners/26b150363d5757da8578fa6a1481585a368f12d4247adfe95837e6ea6c5ab2af-rimg-w1200-h549-gmir.jpg?v=1569691155

Compilers – Week 2

Thank you for your attention!

Contact: sousap@algonquincollege.com

ALGONQUIN COLLEGE

ALGONQUIN COLLEGE

CST8152

Lecture 04

Inside Compilers

Week 3: Inside Compilers

- *Parts of Compilation Process*

L1	L2
A2W01	A2W09
A2W02	A2W10
A2W03	A2W11
A2W04	A2W12
A2W05	A2W13
A2W06	A2W14
A2W07	A2W15

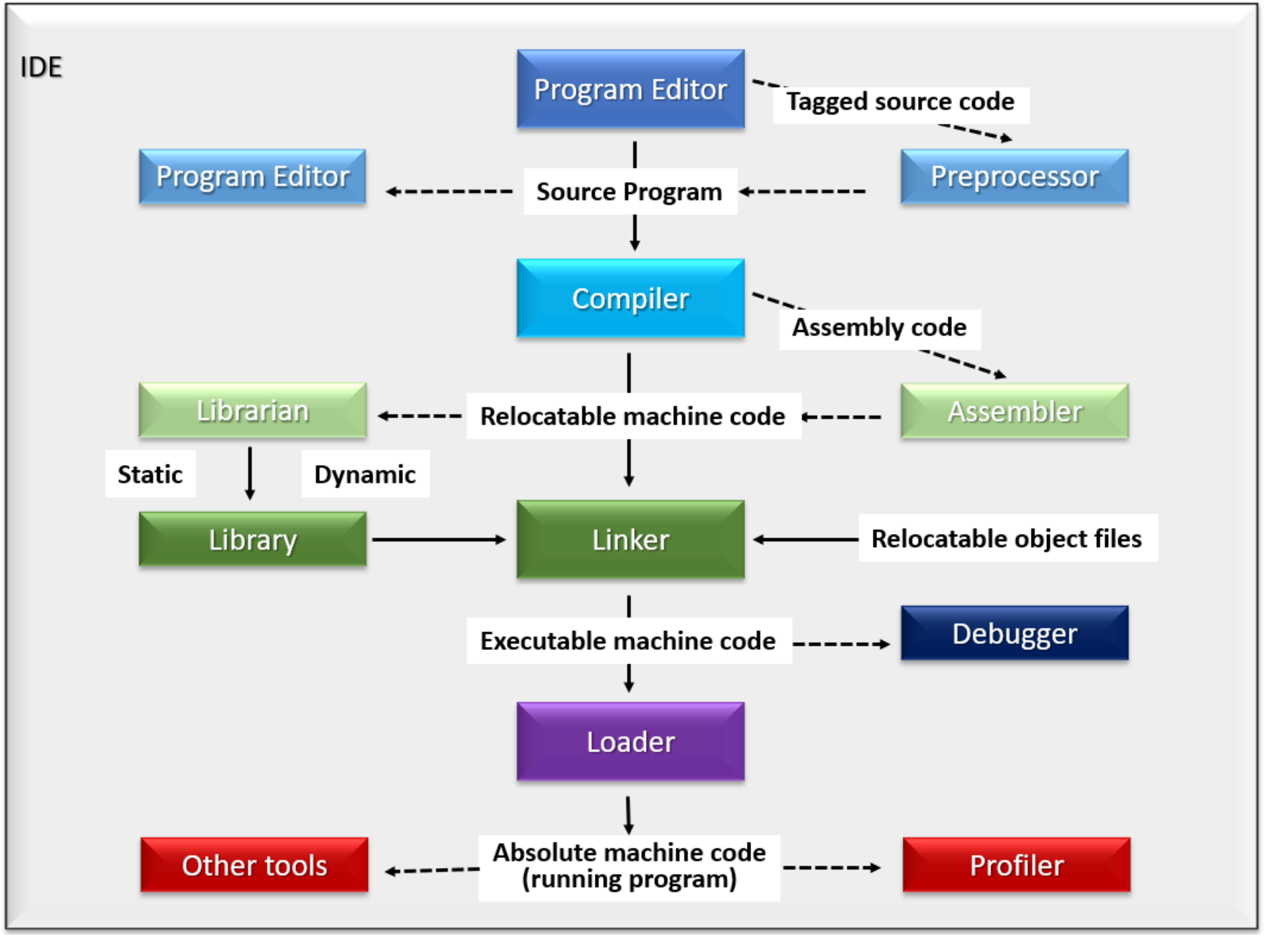
A2W08

ALGONQUIN COLLEGE

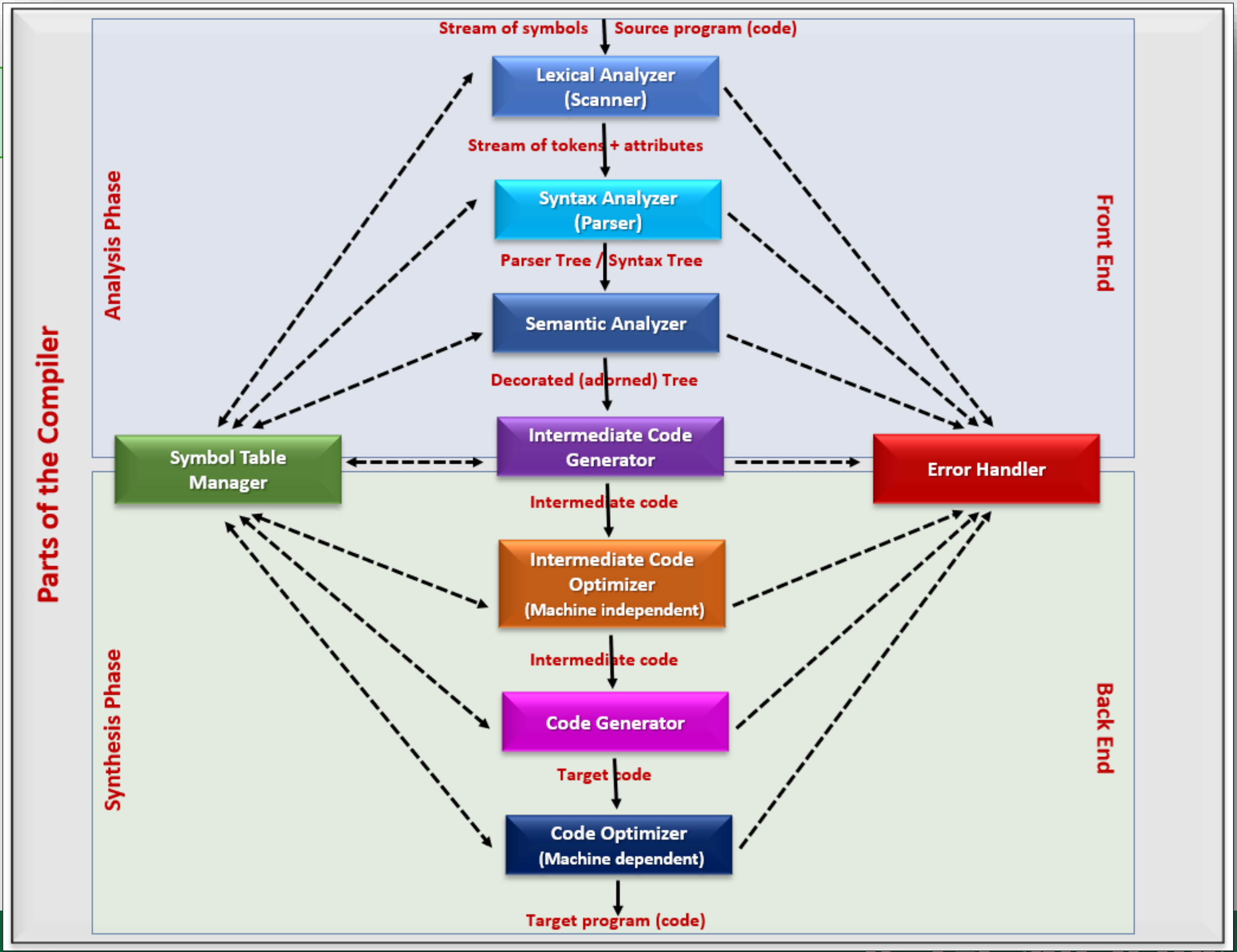
Compilers – Week 3

General View

4.1 - Review


IDE
Program Editor
Tagged source code
Program Editor
Source Program
Preprocessor
Compiler
Assembly code
Librarian
Relocatable machine code
Assembler
Static
Dynamic
Library
Linker
Relocatable object files
Executable machine code
Debugger
Loader
Other tools
Absolute machine code (running program)
Profiler

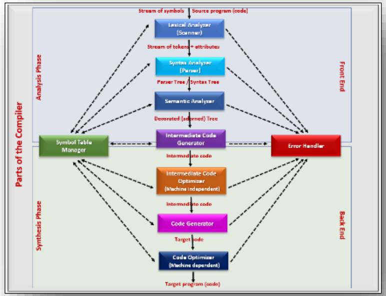
4.1 - Review


Stream of symbols
Source program (code)
Lexical Analyzer (Scanner)
Stream of tokens + attributes
Syntax Analyzer (Parser)
Parser Tree / Syntax Tree
Semantic Analyzer
Decorated (adorned) Tree
Intermediate Code Generator
Intermediate code
Intermediate Code Optimizer (Machine independent)
Intermediate code
Code Generator
Target code
Code Optimizer (Machine dependent)
Target program (code)
Symbol Table Manager
Error Handler
Parts of the Compiler
Analysis Phase
Synthesis Phase
Front End
Back End

4.1. Some concepts

Analysis part (phase)

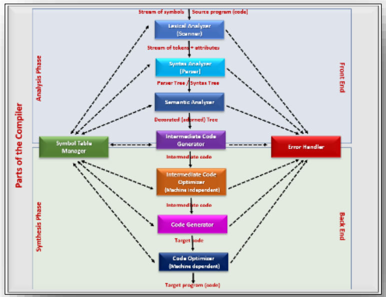
- The first phase of a compiler is called ***lexical analysis*** or ***scanning***.
  - The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
  - For each lexeme, the lexical analyzer produces as output a token of the form: (token - name (code), attribute - value).



4.1. Some concepts

Analysis part (phase)

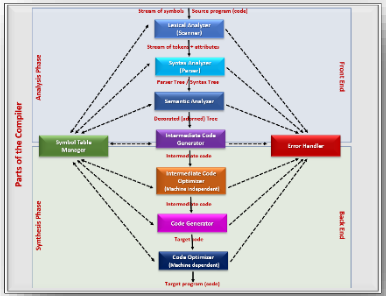
- The second phase of the compiler is ***syntax analysis*** or ***parsing***.
  - The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation called ***parse tree*** that depicts the grammatical structure of the token stream.
  - Typically the parse tree is reduced to another form of representation called ***syntax tree*** in which each interior node represents an operation and the children of the node represent the arguments of the operation.



4.1. Some concepts

Analysis part (phase)

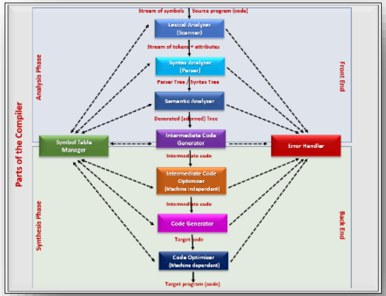
- The ***semantic analyzer*** uses the syntax tree and the information in the ***symbol table*** to check the source program for semantic consistency with the language definition.
    - It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.



4.1. Some concepts

Synthesis part (phase)

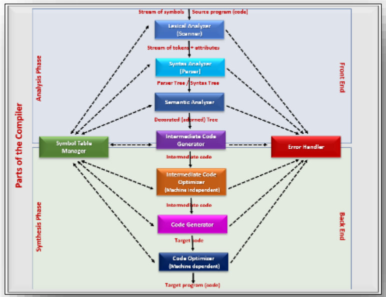
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or ***machine-like intermediate representation***, which we can think of as a program for an *abstract machine*.
    - This intermediate representation should have two important properties:
        - it should be easy to produce and
        - it should be easy to translate into the target machine.
    - Typically, the intermediate representation separates the front end from the back end.



4.1. Some concepts

Synthesis part (phase)

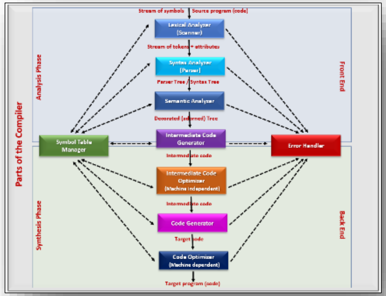
- The **machine-independent code-optimization** phase attempts to improve the intermediate code so that better target code will result.
- There is a great variation in the amount of ***code optimization*** different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase.
  - There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.
- The ***code generator*** takes as input an intermediate representation of the source program and maps it into the target language



4.1. Some concepts

Synthesis part (phase)

- The ***symbol table*** is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The ***error handler*** is responsible to report to the programmer the lexical, syntactical, and the semantic error discovered during the compilation process.
  - It is also responsible to prevent the compiler from producing a target code is an error has been detected.



Compilers – Week 3

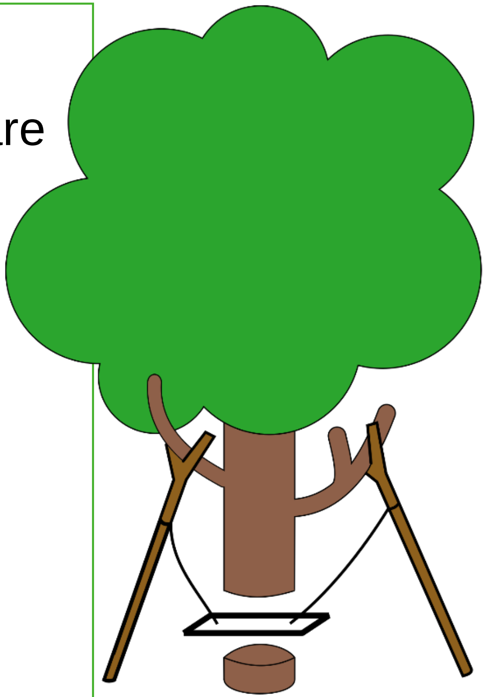
General Purpose Languages

4.2. The “ontological” problem

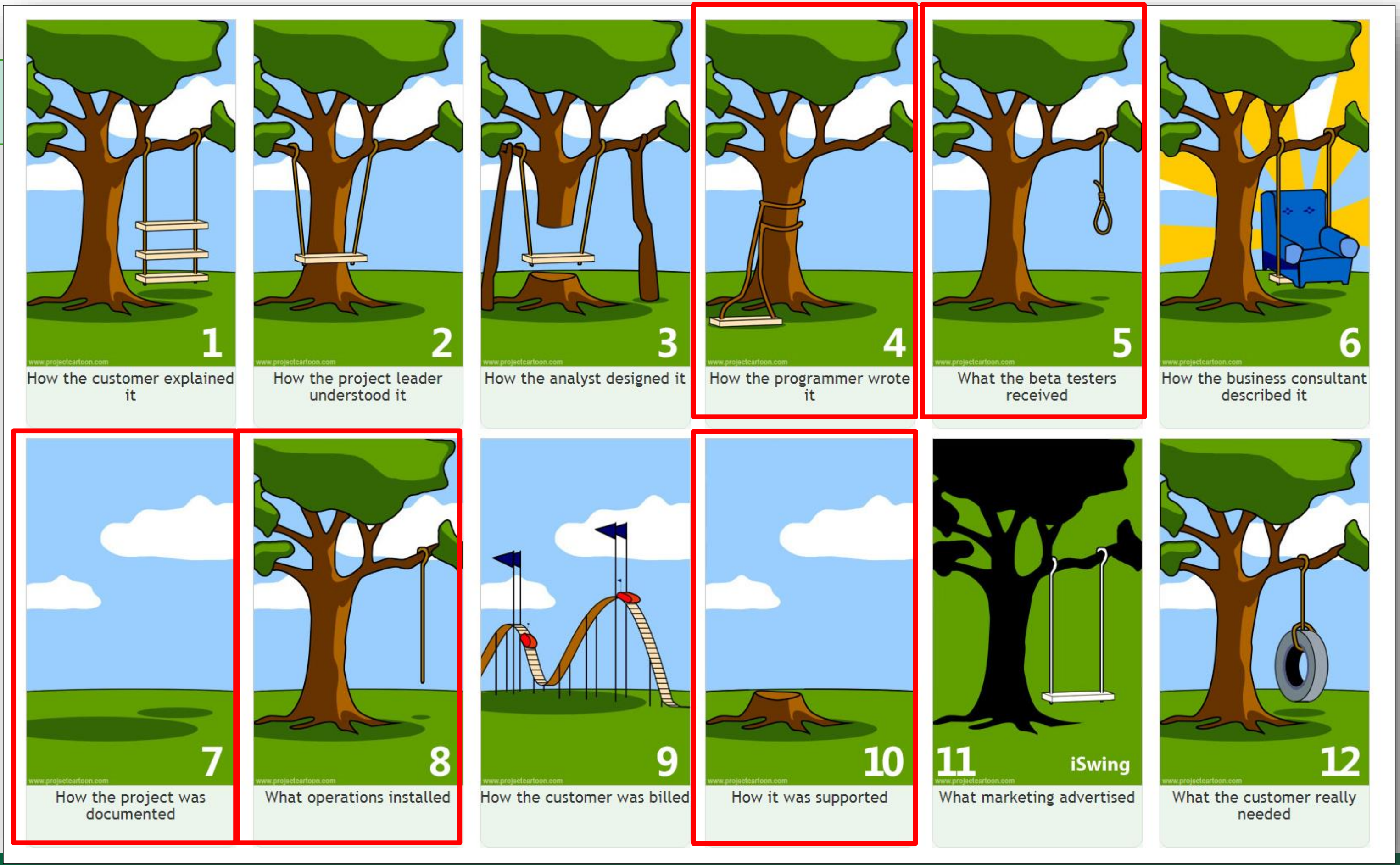
GPL (General Purpose Languages)

- Should be able to create “artefacts” for software development;
- But software needs to attend business needs;
- Business are domain-specific

Note: Part of the objective of SE (Software Engineering) is decrease the “gap” between the idea and the implementation…



I know!


1
www.projectcartoon.com
How the customer explained it
2
How the project leader understood it
3
How the analyst designed it
4
How the programmer wrote it
5
What the beta testers received
6
How the business consultant described it
7
How the project was documented
8
What operations installed
9
How the customer was billed
10
How it was supported
11
iSwing
What marketing advertised
12
What the customer really needed

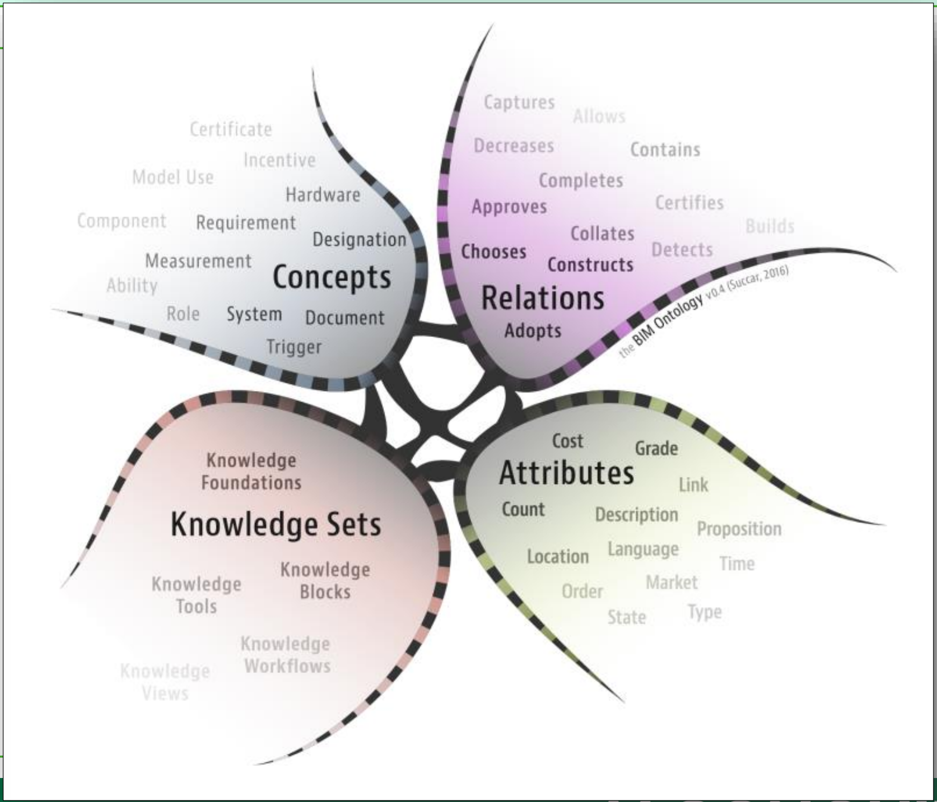
ALGONQUIN COLLEGE

4.2. The “ontological” idea

How to explain languages…

- For a “normal” people;

Note: Even OO is not a real “paradigm” to users, but for programmers.



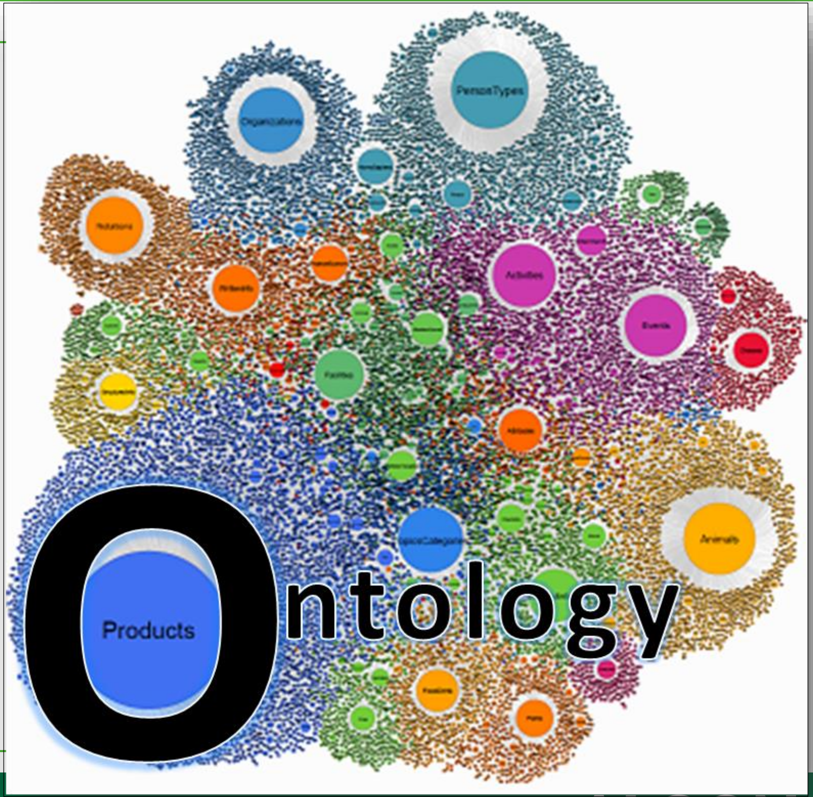
4.2. The "ontological" idea

How to explain languages…

Note: Is it possible to create high-level languages?

Imagine the complexity of concepts and associations required to implement "ontological" languages…

https://www.enterrasolutions.com/wp-content/uploads/2015/03/Ontology-01.png



Compilers – Week 2

Concluding

Review

- *Define the elements of the context of the diagram.*

Source: https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Some Questions

1. *Why to separate analysis from synthesis?*
2. *Explain the purpose of analysis and give one example (different from book);*
3. *Do the same for synthesis phase.*
4. *What is the complexity of GPL?*

ALGONQUIN COLLEGE

Open questions…

- Any doubts / questions?
- How we are until now?

Image URL: https://cdn-az.allevents.in/events3/banners/26b150363d5757da8578fa6a1481585a368f12d4247adfe95837e6ea6c5ab2af-rimg-w1200-h549-gmir.jpg?v=1569691155

Compilers – Week 3

Thank you for your attention!

Contact: sousap@algonquincollege.com

ALGONQUIN COLLEGE

DFA and NFA

DFA

- DFA stands for Deterministic Finite Automata.
- There are a set of states, Q, and a set of symbols in the alphabet Σ.
- There is always one start state, q_0 and one or more accept state(s), F.
- At the beginning, the start state is the state from which tracing the input string starts.
- There is exactly one transition/action to be taken on each symbol of the alphabet.
- The current state is the state at which we end up after a transition is completed.
- A transition may change the current state to a new state or to the current state itself.
- A symbol of the alphabet has only one transition from each state in the DFA.
- DFA is used to recognize if a string belongs to a language based on certain property.
- Tracing a string means taking symbol by symbol and moving from the start state to the final state. The final state is the state which we end up with after the last transition caused by the last symbol in the string.
- The final state may be or may not be an accept state.
- A string is accepted, i.e., belongs to the language, if the final state is one of the accept states.
- A string is rejected, i.e., does not belong to the language, if the final state is not one of the accept states.

NFA

- NFA stands for Non-deterministic Finite Automata.
- There are a set of states, Q, and a set of symbols in the alphabet Σ.
- There is always one start state, q_0 and one or more accept state(s), F.
- At the beginning, the start state is the state from which tracing the input string starts.
- Does not require a transition for each symbol in the alphabet from all states.
    - A symbol may have zero, one, or more transitions from any state.
    - A state may or may not have a transition for each symbol.
- NFA is used to recognize if a string belongs to a language based on certain property.
- Tracing a string means taking symbol by symbol and moving from the start state to the final state. The final state is the state which we end up with after the last transition caused by the last symbol in the string.
- The final state may be or may not be an accept state.
- A string is accepted, i.e., belongs to the language, if the final state is one of the accept states.
- A string is rejected, i.e., does not belong to the language, if the final state is not one of the accept states.

- It is enough to have at least one path from the start state to an accept state in order to recognize that the string belongs to the language. There might be more than one path from the start state to an accept state.
- An epsilon transition, ϵ-transition, may change the current state even without using any symbol in the traced string. This means, if there is an ϵ-transition between two states, say A and B, then if we end up at state A using a symbol upon tracing the string, we could either stay in state A or move to state B.
    - If we have three states: A, B, and C and there are two ϵ-transitions: one is between states A and B and the other is between state B and C. If state A is the current state, after tracing a symbol from the string, then we may stay at state A, or move to state B, or move to state C.

Examples on DFA.

Find the DFA for each of the following languages. The alphabet of all questions is $\sum = \{0, 1\}$

1. The language that only accepts 01.
2. The language accepts either 01 or 10.
3. The language accepts only strings of length 2.
4. The language accepts only strings of lengths >= 2.
5. The language accepts only strings of lengths <= 2.
6. The language accepts only strings that end with 011.
7. The language accepts only strings that start with 011.
8. The language accepts only strings that contain 011 as a substring.
9. The language accepts only strings that have length divisible by 3.
10. The language accepts only strings that have even length.
11. The language accepts only strings that have two 0s.
12. The language accepts only strings that have even number of 0s.
13. The language accepts only strings that start with 0 and end with 1.
14. The language accepts only strings that start and end with different symbols.
15. The language accepts only strings that do not end with 01.
16. The language accepts only strings that have 1 after each 0.
17. The language accepts only strings that have 0^n 1^m where n, m >=1.
18. The language accepts only strings that have the number of 0s is divisible by 3.
19. The language accepts only strings that have even number of 0s and even number of 1s.
20. The language accepts only strings that when interpreted as binary number is divisible by 3.
21. The language accepts only strings that when interpreted as binary number is divisible by 4.

Subset Construction: NFA to DFA Conversion

- This algorithm is used to convert any NFA to a DFA.

- In general, if an NFA has Q states, then the equivalent DFA may have up to 2^Q states.
    - For example, if NFA has two states, then the equivalent DFA may have up to four states.

- The straight forward method for this conversion is to do the following:

    - Create the power set of the NFA states. This means, if the NFA has two states, q1 and q2, then the power set is: ({q1}, {q2}, {q1, q2}, ϕ).

    - If the NFA has three states, q1, q2, and q3, then the power set is: ({q1}, {q2}, {q3}, {q1, q2}, {q1, q3}, {q2, q3}, {q1, q2, q3}, ϕ).

    - Each state in the power set will become a state in the resultant DFA. For example, if the NFA has two states q1 and q2, then the four states of the DFA are: d1 → {q1}, d2 → {q2}, d12 → {q1, q2}, and dϕ → ϕ.

    - For each DFA state, we find the destination state of each transition for each symbol of the alphabet. For example, if Σ = {0, 1}, then we need to find: δ(d1, 0), δ(d1, 1), δ(d2, 0), δ(d2, 1), δ(d12, 0), δ(d12, 1), δ(dϕ, 0), and δ(dϕ, 1).

    - The previous step is based on using the given NFA. For each transition on each symbol, we find all possible states at which this transition may end at including those reachable through the ϵ-transition(s). For example, if we have q1 – a –> q2 – ϵ –> q3 – ϵ –> q4 – ϵ –> q5, then having an 'a' symbol at state q1 will result in the following set {q2, q3, q4, q5}.

    - For composite states in DFA, the final destination given one symbol is the union of all end states of each state of the composite state based on the NFA.

    - For example, for state d12, the final state for a transition on symbol '0' is the union of all states for each of q1 and q2 (using the NFA). The union of all states will be one of the states in the power set that is given a unique DFA state. pay attention to ϵ-transition(s) when finding the possible end states for any transition.

    - Once all destination states for all symbols in the alphabet are found, the DFA can be constructed using the found information.

- The start state of the DFA is the state that represents q0 and its ϵ-closure, where q0 is the start state of the NFA. This means that in order to find the start state in the DFA, we find the ϵ-closure (q0) and add q0. ϵ-closure is the set of all states reachable from q0 via ϵ-transition. In other words, the start state in DFA is the set of the power set that has the start state in NFA and all states that can be reached via ϵ-transition(s) via q0.

- The accept state in the DFA is any state that has the accept state in the NFA. For example, if q1 is an accept state in NFA, then d1, and d12 are two accept states in DFA.

Observations:

1. The method described above is good for NFA that has a few states (Q = 2 ~ 3). For NFA with states of 4 or more, one needs to examine an exponential number of DFA states. For example, if Q (NFA) = 10, then Q (DFA) = 2^{10} = 1024. This is not possible and there should be another way to handle such cases.
2. Most of NFAs (with Q states) can be converted to DFA with states less than 2^{Q}. in the example covered in the class, the NFA has two states and the equivalent DFA has only three states. A reduction was possible because one DFA state was unreachable and hence, it could be eliminated. This leads to a systematic way to find an equivalent DFA of an NFA that has many states, i.e., four or more.

The systematic way is to follow these steps:

- Find the DFA seed state (d0) which is the ϵ-closure of the start state in the NFA.
- For each symbol in the alphabet, we find all transitions from d0. In this step, we must take the ϵ-closure of the answer. This means that we need to consider all states reachable to the state that the symbol takes us including all states reachable via ϵ-transitions. Then, the new set of states is considered as a newly discovered DFA state, e.g. d1.
- Based on the transitions on each symbol for d0, we may discover a new state for the DFA. These newly discovered states are added to the list of the DFA states.
- We repeat the steps carried out for d0 for each of the newly discovered states. In each step, we keep observing if we are getting new DFA states or not. If yes, we add the new state and find ϵ-closure(δ(dn, x)) where dn is the new DFA state and x is one symbol.
- We repeat this until no more new states are discovered.
- The table generated from this algorithm is the transition table of the equivalent DFA which can be programmed easily.