Linking *f*1 and *f*2

Memory Math

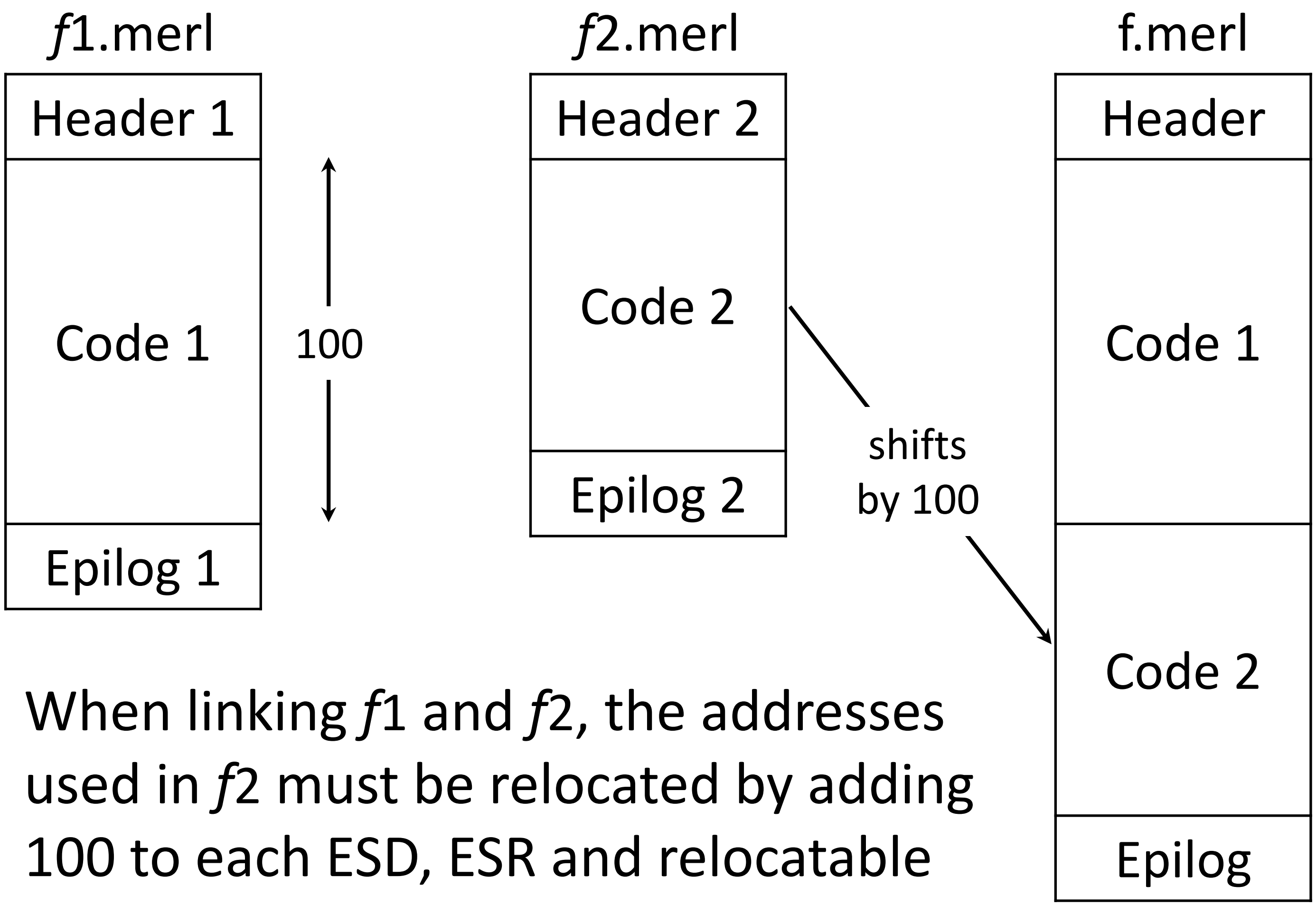
- Because memory locations start at 0 and each word / instruction is 4 bytes, storing data works as follows
  - to store 1 word, i.e. 4 bytes, at address 0x0, locations 0x0 - 0x3 are used and 0x4 is the address of the first free location

X	X	X	X				
0	1	2	3	**4**	5	6	7

  - to add 2 more words, 4 + 8 = 0xC (i.e. 12) bytes, locations 0x0 - 0xB are used and 0xC is the address of the first free location

X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y				
0	1	2	3	**4**	5	6	7	**8**	9	A	B	**C**	D	E	F

Linking *f*1 and *f*2



When linking *f*1 and *f*2, the addresses used in *f*2 must be relocated by adding 100 to each ESD, ESR and relocatable address.

Topic 7 – Regular Languages

Key Ideas

- compiler
- scanner, lexical analyzer, lexer
- regular expressions: union, concatenation, Kleene star
- formal languages: alphabet, words, language

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.
- available (for free, legally) on the web

Creating a Program

Overview

- We now understand enough about assembly language and machine code to be able to convert a few lines of a high level language into their equivalent instructions in MIPS assembly language.
- *Key question: how does a compiler translate a high level language, like C++, into machine code?*
- Hint: it takes several steps.

Creating a Program

Classical Tool Chain

- *Compiler* translates a high level language (such as C++) into an assembly language program (such as MIPS assembly language).
  - You can view the assembly language it generates using the –S option in gcc/g++
- *Assembler* translates an assembly language program into machine code in an object file (e.g MERL or ELF).

The Compiler

What a Compiler Does

- *defining task: a compiler translations a program*
  - *from source language*
  - *to target language*
- typically from a high-level language (e.g. C++) to low-level language (e.g. MIPS assembly)
  - i.e. from a complex (feature rich) language to a simple one
- typically followed automatically by an assembler
  - to generate machine code
- compiling has some similarities with assembling...

The Compiler

Basic Compilation Steps

The *steps in compiling* a program from a high level language to an assembly language program are:

1. *scanning*: create a token sequence (we provided this step for you in Assignments 3 and 4).
2. *syntax analysis*: create a *parse tree* (new)
3. *semantic analysis*: create a symbol table (similar to Assignments 3 and 4) and *type checking* (new)
4. *code generation*: similar, but more complicated for a compiler (as compared to an assembler)

The Compiler

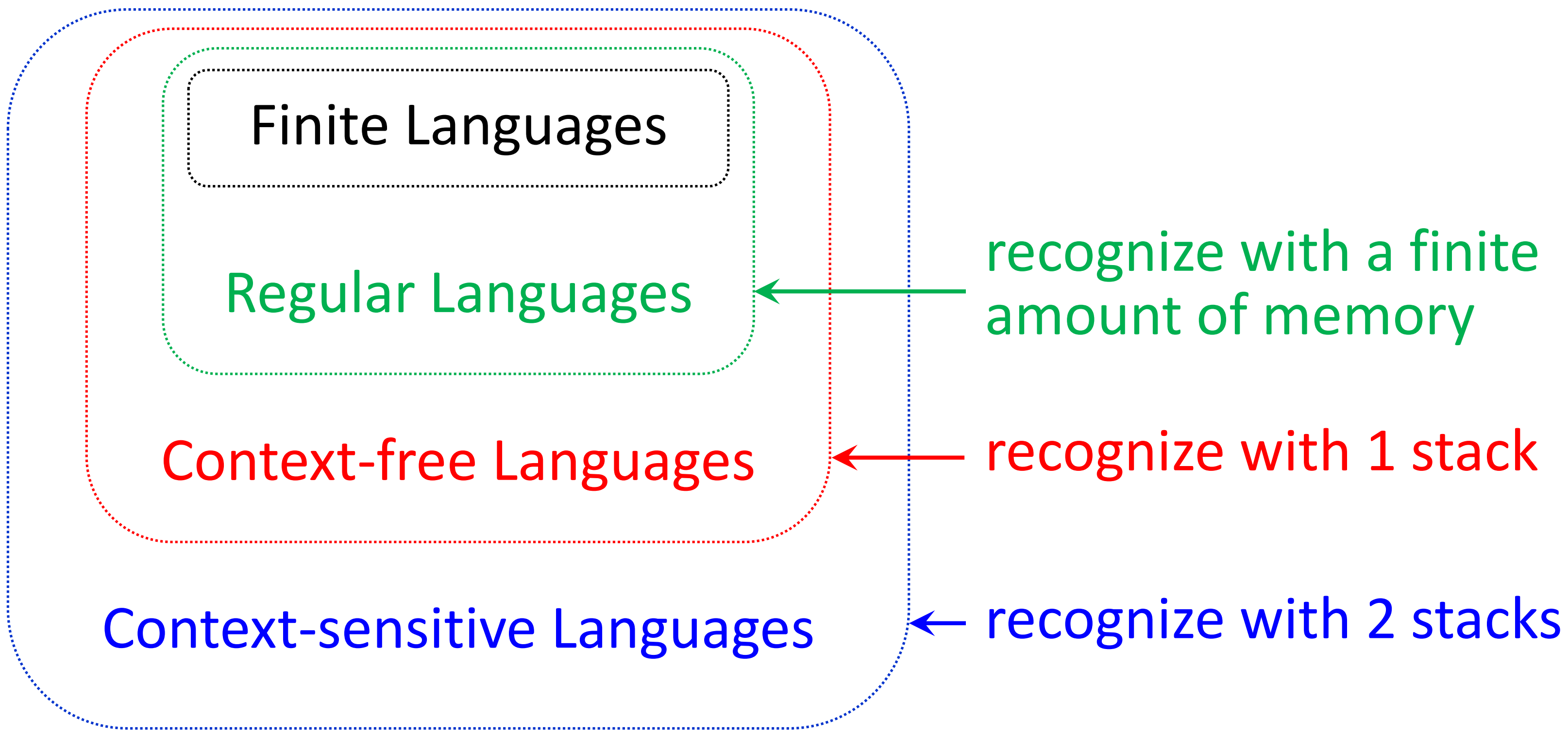
Basic Compilation Steps

- The goal of each of these steps is to *find increasingly more sophisticated errors* in a program.
- And if the program does have an error, then identify
  - the likely source of the error
  - how to fix it
- General approach: define an *increasingly more sophisticated set of languages,* i.e. the Chomsky Hierarchy, that can catch increasing more sophisticated types of errors.

Caution: no compiler can find all errors.

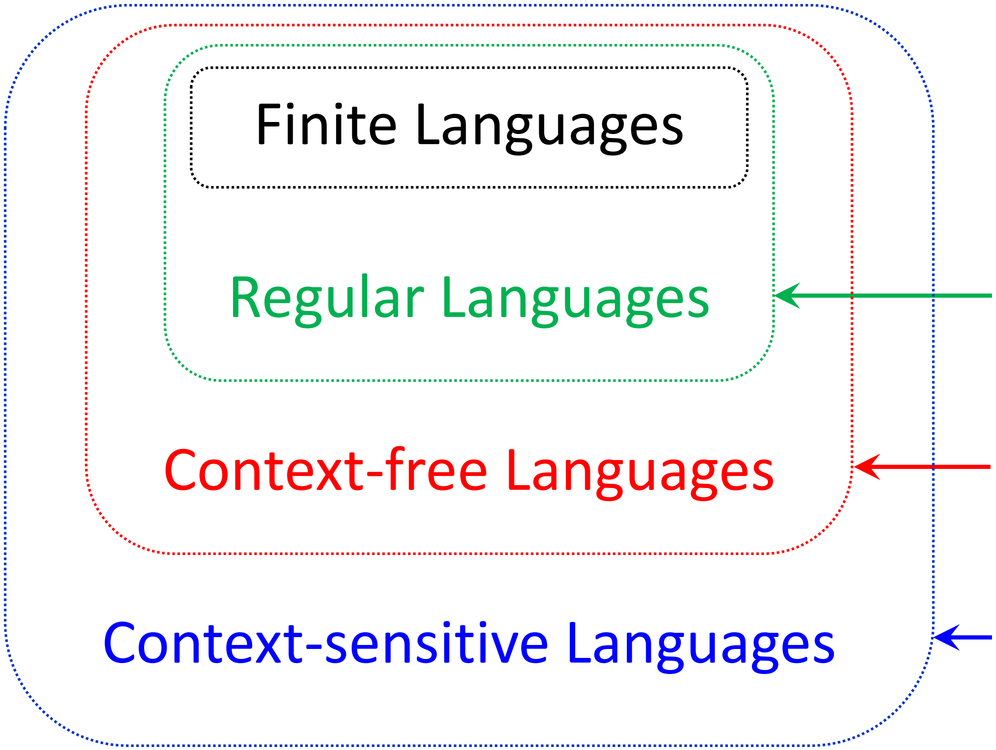
The Compiler

Chomsky Hierarchy



The Compiler

Chomsky Hierarchy



Steps in Compiling

1. lexical analysis: find each lexical element
2. check the syntax
3. check the semantics
4. code generation

Step 1: Scanning

What is a *scanner*?

- a.k.a. a *lexical analyzer* or *lexer*
- Recall: A scanner answers the questions: What are the keywords, names, operators, etc, in the code?

```
int  maxEntry (int *anArray, int  numRows) {
 // return the maximum entry in anArray
 int i,  answer = 0;
  for (i=0; i<numRows; i++) {
   answer += anArray[i];
  }
  return answer;
}
```

Step 1: Scanning

Tokens

- also called *lexical analysis*
- convert code into a stream of (token types, token value) pairs
- need more tokens for a high level language than for assembly language, e.g.
  - *keyword*: int float if for while return ...
  - *operator*: + - * / = += < <= > >= == != ...
  - *constant*: 0 2.1 "Hi" ...
  - *delimiter*: ( ) { } [ ] , ; ...
  - *name*: maxEntry anArray numRows i answer ...

Step 1: Scanning

Scanner Input

```
int  maxEntry (int *anArray, int  numRows) {
    // return the maximum entry in anArray
    etc.
```

Scanner Output

- (INT, 'int')
- (ID, 'maxEntry')
- (LPAREN, '(')
- (INT, 'int')
- (STAR, '*')
- (ID, 'anArray')
- (COMMA, ',')
- (INT, 'int')
- (ID, 'numRows')

  etc.

Step 1: Scanning

Scanning

- *keywords*
  - easy to recognize
  - there are a fixed number of them, roughly 10 in WLP4 (CS241's Waterloo Language Plus Pointers Plus Procedures)
  - there is *never any ambiguity* about them
  - you cannot have a variable named *for* in C++
- *delimiters and operators*
  - easy to recognize
  - there are a fixed number of them
  - *some ambiguity*: ( ) are used as delimiters for function arguments and in a *for* loop.

Step 1: Scanning

Scanning

- *constants and names*
  - harder to recognize: variable length
  - need some sort of pattern matching
  - must determine when this token ends and the next one begins
  - ambiguity: if the first three characters are '241',

    is this an integer or a float?
  - there are an infinite number of possible names and constants in a typical programming language
- *Challenge 1:* how to *specify* all the elements in the infinite set of valid tokens for CS241's WLP4, C++, Racket, etc.

Scanning Background

Task

- *Challenge 2:* clearly and unambiguously *recognize* all the tokens in a computer language, say WLP4.

Complications

- names and constants have variable length
- some tokens, such as ‘(‘, mean different things in different contexts
- there are many types of names: function names, function arguments, global variables, local variables
  - have to be able to recognize these different types
- We will use a formal language.

Formal Languages

Why Formal Languages?

Goal: give a precise specification of a language

- describe (specify) a computer language, such as C++,
- in such a way that it is possible to tell if input (i.e. a program) meets the specification
- in an automated fashion (i.e. a computer program).

Why do we need a formal (i.e. mathematical) way?

- as a means of communication
- to determine the expressive power and limitations of the language
- to guide how to make the software

Scanning Background

Approach

- use *regular expressions* to describe our language
- then use a *lexer generator*
  - converts our language description into an efficient program for recognizing the tokens
  - examples of lexers are: lex, flex, ANTLR
- Lexers are an example of programs called *finite automata* (more about these later).
- But first, what is a *regular expression*?
- Answer: a *precise way of describing a set of strings* (think programs or sequence of characters)
- where a string is a finite sequence of characters over some alphabet

Regular Expressions

Linux

- For those of you who use Linux, you use these all the time
  - `ls A2*.asm`
  - list all the files that start with 'A2' and end with '.asm'

In general

- recall a *string* is a sequence of characters over a finite alphabet
- *We want to be able to define set of strings over an finite alphabet Σ*, i.e. rooted in set theory
- For programming languages our alphabet will generally be some subset of the ASCII characters (i.e. the characters on an American keyboard).

Regular Expressions: Constants

Constants

- similar to the empty set, Ø, which has no elements we have the empty string, ε, which has no characters in it.
- literal character: *a* in Σ
  - all the individual characters in the alphabet
  - the alphabet is finite but the language may be infinite
- This defines the single elements, but *how do we combine them?*

Regular Expressions: Basic Operations

The 3 Operations for Building up Languages

1. *Union*

R | S is the union of set R and S, i.e. R U S

- if R and S are regular languages, then so is R U S
- if R = {dog, cat} and S = {cow, pig}, then R | S = {dog, cat, cow, pig}
- if R and S are regular languages, then so is R | S

2. *Concatenation*

RS = { αβ : α in R and β in S}

- take a word from R and combine it with a word from S
- if R = {grey, blue} and S = {jay, whale}, then RS = {greyjay, greywhale, bluejay, bluewhale}

Regular Expressions: Basic Operations

The 3 Ways of Building up Languages

2. *Concatenation* (continued…)

- concatenation with the empty string, ε, does nothing, i.e. $\alpha\varepsilon = \alpha$
- ε is the identity element under concatenation, like 0 is for integer addition, i.e. $0 + x = x$
- if R = {dog, cat} and S = {fish, ε}, then RS = {dog, cat, dogfish, catfish}
- if R and S are regular languages, then so is RS.

Regular Expressions: Basic Operations

The 3 Ways of Building up Languages

3. *Repetition* (a.k.a. *Kleene star*)

R* = smallest superset of R containing ε and closed under concatenation

- all possible combinations of the elements in R
- if R = {a} then R* = { ε, a, aa, aaa, aaaa, aaaaa, ... }
  i.e. any finite sequence of a's including no a's
- if R = {0, 1} then R* = { ε, 0, 1, 00, 01, 10, 11, 000, 001, ... }
  i.e. any finite sequence of 0's and 1's including ε
- in both these cases the size of the language R, i.e. |R|, is infinite.
- if R is regular languages, then so is R*

Regular Expressions: Basic Operations

The 3 Ways of Building up Languages

3. Repetition (a.k.a. *Kleene star*)

If R is a language, can talk about R^0, R^1, R^2, R^3, etc.

- e.g. if R = {0, 1} then

R^0 = { ε }, i.e. the empty string

R^1 = {0, 1}, all single elements

R^2 = {00, 01, 10, 11 }, all pairs of elements

R^3 = { 000, 001, 010, 011, 100, 101, 110, 111, }, all triplets

Regular Expressions: Examples

A Finite Language

- *Alphabet* Σ = { a, b }

  is a set of characters

  i.e. there are only two characters in this alphabet

- *Words* (a.k.a. strings or sentences) are finite sequences of characters from the alphabet

  e.g. ‘a’, ‘b’, ‘ba’ ‘abba’ ‘bababa’

- A *language* is a set of words over some alphabet

  e.g. $\mathcal{L}$ = {‘a’, ‘b’, ‘ba’, ‘abba’, ‘bababa’ }

- Languages can be finite or infinite

  e.g. $|\mathcal{L}|$ = 5 means the language $\mathcal{L}$ has five words in it.

Regular Expressions: Examples

Some Finite Languages

- the empty set Ø or { }
- { ε }

  the language that consists of the empty string
- ε* = { ε }

  Kleene star of the empty string is just the empty string
- { while }

  the singleton set consisting of the word *while*
- (h|c)at = { hat, cat }
- (a|b)(c|d) = { ac, ad, bc, bd }

Regular Expressions: Examples

Some Infinite Languages

- a* = { ε, a, aa, aaa, ... }

  i.e. any finite sequence of a’s including no a’s
- {a, b}* = { ε, a, b, aa, ab, ba, bb, aaa, aab ... }

  i.e. any finite sequence of a’s and b’s including the empty string
- b|a* = { b, ε, a, aa, aaa, ... }

  b or any finite sequence of a’s including no a’s
- (0|1)* = finite binary numbers, plus empty string

Regular Expressions: Linux Tools

Regular Expressions in Linux

- egrep
  - search regular expressions in text files
- sed
  - stream editor for transforming text files
- awk
  - pattern scanning and processing language
- make
  - software building utility
- *You don't have to know about any of these tools.*

Regular Expressions: Example

Regular Expressions in Linux

- search for all occurrences of a name in a text
- different spelling for Georg Friedrich Händel:
  - Händel
  - Haendel
  - Handel
  - Hendel

```
egrep "[Hh](ae|a|e|ä)ndel" ex.txt
```

Regular Expressions: Examples

Regular Expressions in our C++ like Language

- *Keywords*

  int | float |for | while | return | if | else

  I’ve listed 7 of them, but there are many more.

- *Operators (same for delimiters)*

  +| - | * | / | = | += | < | <= | > | >= | == | !=

  I listed 12 of them, but there are many more.

- *Names*

  must start with a letter or underscore, then any finite combination of letters, underscores, numbers and

  [a-zA-Z_][a-zA-Z_0-9]*

Regular Expressions: Issues

Regular Expressions in our C++ like Language

- would also have to specify the format of integers, floats, string constants.
- *conflicting rules*: need precedence rules
  - does a|ab* mean (a|(ab))* or a|(a(b*)).
  - use order of rules
- usually use greedy approach (produce longest possible match)
  - for 123.45

    if I stop at 3, I get the integer 123, but I can continue

    if I stop at 5, I get the float 123.45 and the longest match)

Regular Expressions vs. Finite Languages

Regular Expressions in MIPS Assembly Language

Is it a finite language? Is a regular language?

$\mathcal{L}_1$ = {\$0, \$1, … \$31} the set of valid MIPS registers.

$\mathcal{L}_2$ = the set of all valid MIPS labels

$\mathcal{L}_3$ = the set of all valid MIPS offsets (for lw and sw)

$\mathcal{L}_4$ = is a of valid line for Assignment 3 Problem 4 (can use labels as operands for the `.word` directive)

$\mathcal{L}_5$ = the set of all valid MIPS assembly language programs

Recognizing A Regular Expression

Task

- clearly and unambiguously be able to recognize all the tokens in a computer language

Approach

- once we've *specified* our programming language with regular expressions...
- we need to *recognize* it with: non-deterministic finite automata (NFA)
- but first Deterministic Finite Automata...