CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Spring 2023

LAST WEEK

Overview of a Compiler:

- Front End
- Back End

TODAY

- Role of the Lexer (also called Scanner)
- Topics: Lexical Analysis, Lexical Tokens
- Regular Expressions
- Lexical specification using regular expressions

“To translate a program from one language into another, a compiler must pull it apart and understand its structure and meaning, then put it back together in a different way.”

Front End

Lexical Analysis: break input into <u>tokens</u>

Syntactic Analysis: parse phrase structure of source code

Semantic Analysis: check program's meaning?

Lexical Analysis

Lexical Analyzer (LA)

Input: stream of characters

Output: stream of "tokens" (recognize in the source code all names/identifies, keywords, punctuation marks)

- Usually will discard any white space and comments between the tokens.

- LA is not very complicated
- Formalisms and tools useful in *lexing* phase can also be used in *parsing* phase.

Lexical Analyzer (LA)

Goal: Divide up code into tokens

1. Place "dividers" between different tokens
2. Recognize the token type or class of each token

```
if (i == j)
  z = 0;
else
  z = 1;
```

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Token Classes

Each token class correspond to a set of strings.

Examples:

- **ID**
  class description: *strings of letters or digits starting with a letter*
  examples: `foo` `n14`
- **NUM** *non-empty string of digits*
  `73` `0` `00` `082`

Other classes are by themselves, (*singletons*):

- **IF** `if`
- **LPAREN** (

Also defining the token class:

- **WHITESPACE** *a non-empty sequence of blanks, newlines, and tabs*

Dataflow

- Will classify program substrings according to their token type/class.
- Communicate tokens to parser.

General Form of Token: <class, string>

- String part is often called a lexeme.
- *Examples of tokens with their lexemes:*
  <Id, "foo"> <Op, "="> <Int, "42">

How to Implement Lexical Analysis?

- *Goal:* partition the source program (a very long string) into its token substrings
- *Idea:* recognize one token at a time
- *Implementation:* read string left-to-right
- *How:* eventually will use a finite automaton

Example

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Defined Token Classes:

Operators
Whitespace
Keywords
Identifiers
Numbers
… and some singleton classes: ( ) ; =

Example

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Lookahead may be required to decide where one token ends and the next token begins.

Q: Where is lookahead necessary?

Ideally lookahead is not too costly from a computational perspective.

- But still would like to *minimize* the amount of lookahead required.
- Depending on the language specification (what language you are compiling for), minimizing lookahead may be easier or more difficult.

FORTRAN Example

```
      sum = 0
      do 5 i = 1,25
        sum = sum + i
        write(*,*) 'i = ', i
        write(*,*) 'sum = ', sum
   5  continue // loop back
```

- do loop is a strange looking instruction!
- *Meaning:* iterate using the counting variable until the label 5 is reached.
- Loops 25 times, from 1 to 25, stepping by 1.
- BUT, whitespace is insignificant in FORTRAN (like Java). Could revise program by eliminating whitespace and it would be exactly the same.
- `do5i=1,25` This instruction is also perfectly valid.

`do 5 i = 1,25` vs. `do 5 i = 1.25`

- Very different meanings! What's the problem?

Loop Assignment statement w/ variable and float “`do5i = 1.25`”

Regular Languages

- used to specify the lexical structure of programming languages
- lexical structure: <u>set</u> of token classes/types
- each token class contains some <u>set of strings</u>
- will use <u>regular languages</u> to specify which set of strings belongs to a token class

Formalisms

- A formal language has an alphabet Σ (sigma): a set of characters.
- Language is a set of strings drawn from that alphabet.
- To define regular languages, we generally use regular expressions.
  - each regular expression denotes a *set*

Regular Expressions are built *recursively* out of smaller regular expressions

Two "base case" building rules:

1. Singular Character

Example: `'d' = {"d"}`

The regular expression that is the single character d denotes
the *language* containing one string, which is the single character d.

Recall: <u>A language is a set of strings.</u> (A string is a finite sequence of symbols; the symbols are from a finite alphabet.)

Regular Expressions are built *recursively* out of smaller regular expressions

Two "base case" building rules:

2. Epsilon

Example: ε = {""}

Represents the language that contains a single string, the *empty string*.

Side note: This is not the empty language, which is the empty set of strings.
$\varepsilon \neq \emptyset$

Regular Expressions are built *recursively* out of smaller regular expressions

Three “induction” building rules:

3. Union / Alternation

Example: `A|B = {a|a ∈ A} ∪ {b|b ∈ B}`

- "regex A pipe regex B”

Corresponds to the *union* of the languages in A and B.

- The set of strings, each string a that is in the language of A, union with each string b that is in the language of B.

Regular Expressions are built *recursively* out of smaller regular expressions

Three "induction" building rules:

4. Concatenation

Just like string concatenation.

Example: $A \cdot B = \{ ab \mid a \in A \wedge b \in B \}$

Given two languages (or regular expressions denoting the languages), A and B, the concatenation is equal to all the strings where a is drawn from A and b is drawn from B.

- *Cross-product operation:* choose string from A, and choose string from B, in all possible ways.

Regular Expressions are built *recursively* out of smaller regular expressions

Three “induction” building rules:

5. Repetition / Iteration

Example: $M^* = \bigcup_{i \geq 0} A^i$
(Pronounced "M star".) Also known as the Kleene closure.

Equal to the union of $i \geq 0$, of A^i (to the i^{th} power).

- $A^i \rightarrow A$ concatenated with itself i times. $\underbrace{A \ldots A}_{i \text{ times}}$

- Possible that $i = 0, A^0 = \varepsilon$: language of the empty string.
- Empty strings are always an element of the Kleene closure.

- These 5 cases define the construction of regular expressions over some given alphabet Σ.

Examples

$\Sigma=\{0,1\}$

- 1^*
- $(1|0)\cdot 1$
- $0^*|1^*$
- $(0|1)^*$

More Examples in [Appel]

Multiple regular expressions can denote the same set.

Not necessarily only one unique way to write a language.

Example: $(1|0)\cdot 1 = (0|1)\cdot 1$

These two regular languages are <u>equivalent</u>.

Another Example

$\Sigma=\{0,1\}$

Regular Language: $(0|1)^*\cdot 1\cdot (0|1)^*$

Which of the below *languages* are *equivalent*?

A. $(1|0)^*\cdot 1\cdot (1|0)^*$
B. $(0|1)^*\cdot (1\cdot 0|1\cdot 1|1)\cdot (0|1)^*$
C. $(0\cdot 1|1\cdot 1)^*\cdot (0|1)^*$
D. $(0|1)^*\cdot (0|1)\cdot (0|1)^*$

Backtracking to Formal Languages

- A formal language has an alphabet Σ.
- Language is a set of strings drawn from that alphabet.

- *Example:* Σ = ASCII

- C language: set of all strings that constitute legal C programs.
  - infinite, but well defined
- Language of C reserved words: set of all alphabetic strings that cannot be used as identifiers.

Writing Regular Expressions for Token Classes

Example: Regular Expressions for Keyboards

```
if

else

public
```

i" · "f" | "e" · "l" · "s" · "e" | "p" · "u" …

Inconvenient Notation
Some Abbreviations:

- No dot operator: "i" · "f" ≡ "if”
- Range:[abcd] ≡ (a|b|c|d) ≡ [a-d]
- [b-gM-Qkr] ≡ [bcdefgMNOPQkr]
- Zero or one: M? ≡ (M | ε)
- One or more <u>(Positive Closure):</u> $M^+ \equiv (MM^*)$
  - *Side Notes:* These extensions do not extend the power of regular expressions.
  - Operators ? and + not really necessary.
  - Only need Kleene closure, alternation, concatenation, …

More Regex Examples for Token Classes

- **ID** - identifier: sequence of letters or digits starting w/ a letter.
- **INT** - integer: a *non-empty* string of digits.
- **REAL** - w/ decimal pt.: (example values: 3.14529, 4., .12)
- **WHITESPACE** - a non-empty sequence of blanks, newlines, and tabs

Lexical Specification

<u>Lexical specification:</u> defines the (1) token classes and their (2) regular languages.

- **Lexical specifications should be complete.**
- We want to always match some initial substring of the input.
- Usually will include rule that matches any single character.
- What if nothing matches?

Example

```
if                                 IF
[a-z][a-z0-9]*                     ID
[0-9]+                             NUM
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)   REAL
("--"[a-z]*"\n")|(" "|"\n"|"\t")+     no token, just whitespace
.                                  error
```

Notice that rules are ambiguous.

Which rule/regex should match `if8`?

Disambiguation

- Used by lexing tools: *Lex, JavaCC, SableCC*, among others...

1. Longest match: the next token will be the longest prefix that can match any regex.
2. Rule priority: if there is a tie, the highest regex takes precedence. (So, order of rules does matter.)

Next Step

... an *implementation* so that we can use regular expressions to match lexical tokens on the source code.

Practice Problem Using Longest Match and Rule Priority

Σ={a,b,c}

Given the string: `abbbaacc`

What tokenization will the following lexical specifications produce?

```
Token Class     Regex
=====================
     A            c*
     B            b+
     C            ab
     D            ac*
```

Lexical Specification A

```
Token Class     Regex
=====================
     A            b+
     B            ab*
     C            ac*
```

Lexical Specification B