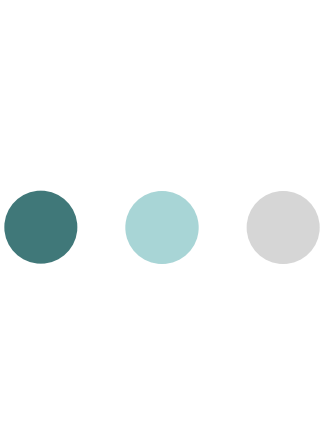


CS4430 – Compilers I

Dr William Harrison

Spring 2017

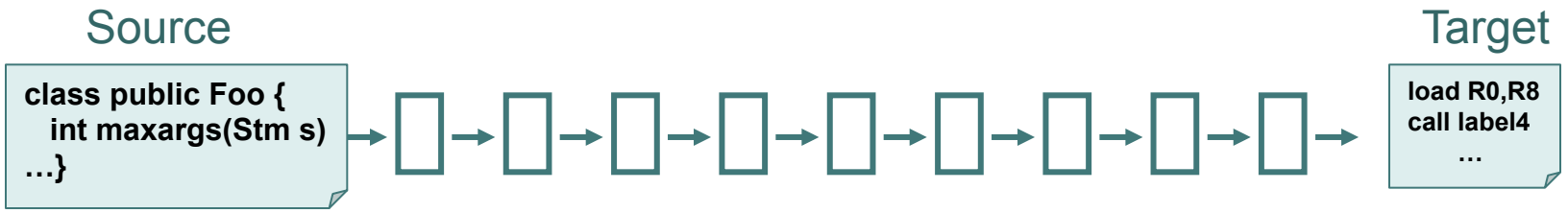
Lexical Analysis

HarrisonWL@missouri.edu

Today's Lecture

- **Begin discussion of "front-end"**
  - I.e., the early phases of the compiler
  - In particular, "lexer"
- **Approach**
  - Start with really simple & concrete example
  - Consider the underlying theory
  - Learn some tools ("lex", "ScanGen", etc.)

Traditional Compiler Structure

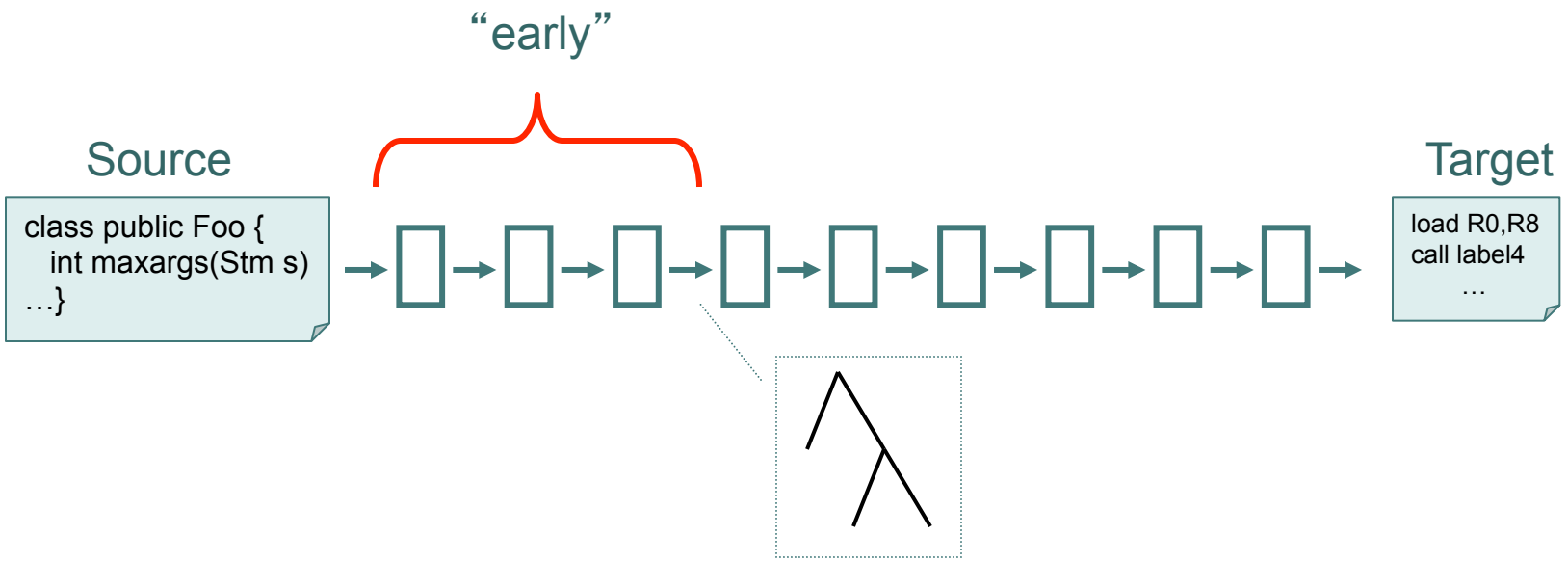


Compilers have “phases”:

- each phase has an input and an output
- each phase transforms its input code into output code
- they are typically classified into “early,” “middle,” and “late” phases

  which accomplish different kinds of transformations

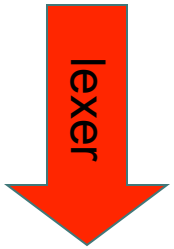
Compiler phases



- early phases transform input sequence into tree representation (AST)
- ensure that input stream is, indeed, a program in the source language
- lexing, parsing, type-checking

What a lexer does

ascii form



symbolic form

class | public | name(“Foo”) | left-brack | type-int | …

Key Concept: regular expressions

How do we “lex” in C?

- I.e., if we want to sit down and write a C program that “just does it”, what will it look like?
  - Introduce “Micro” programming language
    - allows us to “get our feet wet” with compiler issues
    - small, quick, & easy to “digest”
- FYI, “lexing” is also called “scanning”

Informal Micro

- **Data**: only integers
- **Declarations**: there are none
  - all declarations are implicit
  - just use variables without declaring them
    - e.g., like BASIC
- **Literals** (i.e., constants) are strings of digits
- **Statements**:
  - ID := Expression, read(list of IDs), write(list of IDs)
  - all statements end with “;”
- **Programs**: begin and end with “begin” and “end”

Informal Micro

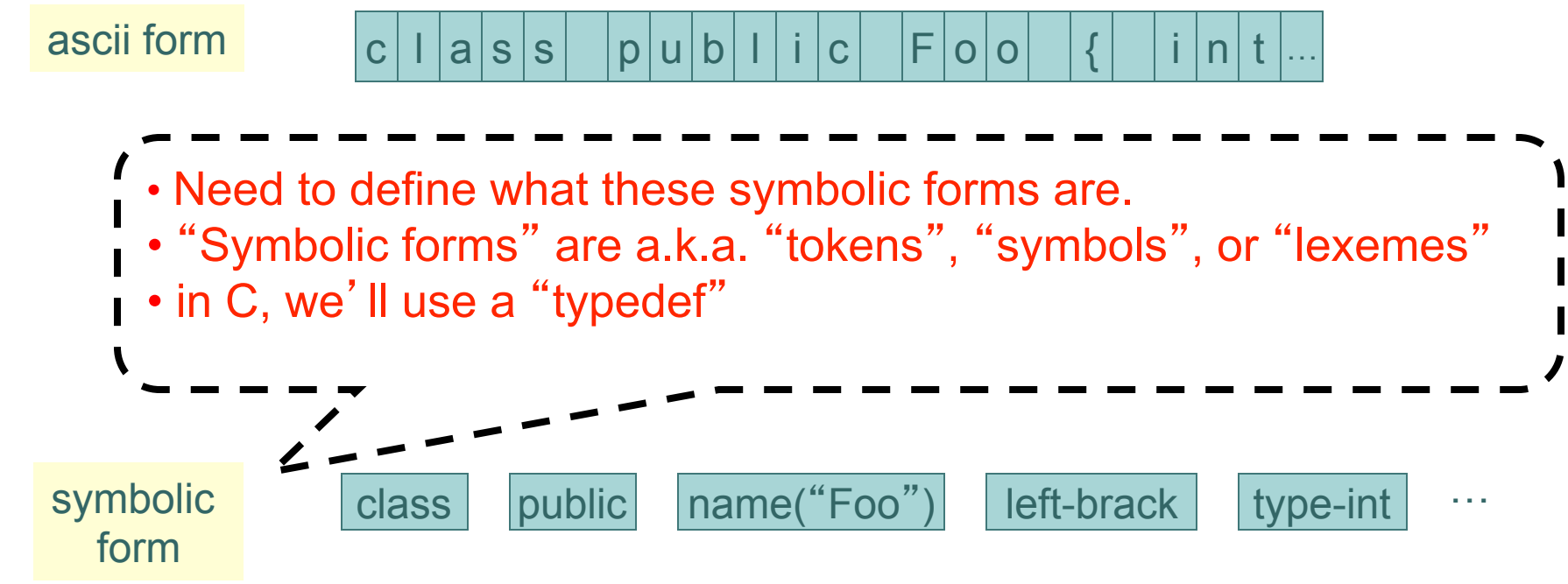
- **Data**: only integers
- **Declarations**: there are none
  - all decl... ons are ...licit
  - just use va... ...n
    - e.g. ...
- **Literals** (... ...gits
- **Statement...
  - ID := ...pression ... (list of IDs)
  - all statements ... ;
- **Programs**: begin an... ...nd w... "begin" and "end"

We' d like a more precise description of language syntax, but that will have to wait

Example: Micro programs

```
begin
  x := 7 + y;
  read(y,z);
end
```

What a lexer does



Key Concept: regular expressions

Tokens for Micro

Micro Source

```
begin
  x := 7 + y;
  read(y,z);
end
```

C tokens

```
typedef enum token_types {
    BEGIN, END, READ, WRITE,
    ID, INTLITERAL,
    LPAREN,RPAREN,SEMICOLON,
    COMMA,ASSIGNOP,
    PLUSOP,MINUSOP,SCANEOF
 } token;
```

Lexing Micro

ascii

“begin\n x:=7+y;\n read(y,z);\n end”

token stream

BEGIN ID ASSIGNOP INTLITERAL PLUSOP ID READ LPAREN…

The problem: translate ascii string into token stream

Ultra-quick review of C

```
#include <stdio.h>

main() {
        printf("hello, world\n");
}
```

```
getchar()          /* returns a character from standard input */
ungetc(c, stdin)   /* puts a character back on the “front” of standard input*/
isalpha(c)         /* true if c is in a-z or A-Z */
```

Remember: single “=“ is assignment, and “==“ is equality test

Using a lexer

Lexer is generally a procedure which returns a single token per call

ascii

“begin\n x:=7+y;\n read(y,z);\n end”

token stream

BEGIN ID ASSIGNOP INTLITERAL PLUSOP ID READ LPAREN…

```
scanner(); /* returns BEGIN */
scanner(); /* returns ID */
scanner(); /* returns ASSIGNOP */
      …
```

Scanner for identifiers

```
 ...includes & declarations...
main() {
        while ((in_char = getchar()) != EOF) {
            if (isspace(in_char)) {
                /* if it's "whitespace", just consume it and go on */
            }
            else if (isalpha(in_char)) {
                /* starts with a char, then it's an identifier */
            } else {
                /* don't recognise in_char --- then it's an error */
            }
        }

}
```

N.b., this is a standalone version

Scanner for identifiers

```
...includes & declarations...
main() {
        while ((in_char = getchar()) != EOF) {
            if (isspace(in_char)) {
                    continue;
            }
            else if (isalpha(in_char)) {
                    for (c = getchar(); isalnum(c) || c == '_'; c = getchar());
                    ungetc(c, stdin);
                    printf("ID ");
                    /*return ID;*/
            } else {
                   printf("argh!\n"); exit(-1);
            }
        }

}
```

Filling in the blanks

- Adding cases
  - mostly simple
    - “`...else if (in_char == ‘;‘) then return SEMICOLON`”
- Distinguishing keywords from variables
  - e.g., “begin” is special, etc.
  - use another helper procedure “check_reserved()”
- Buffering identifiers/literals
  - i.e., one wants to know actual variable names, integer constants
- Better error handling
  - using “error(-1)” just crashes the program.

“Pros & Cons” of this approach

- Pro: Straightforward
  - just roll up your sleeves and start programming
- Con: too concrete
  - it’s hard to tell what the lexer program is doing without knowing a lot of detail about C and inspecting the code closely
- Con: it’s ad hoc
  - it doesn’t take advantage of any engineering experience from over the past 40 years
- Con: error-prone
  - humans don’t do well with “fiddly” details

What we'd like – as much automated support as possible

- Most parts of a front-end are generated rather than written by hand
  - front-end issues are *quite* well-understood
- Tools for lexers: lex, ScanGen,…
- Tools for parsers: yacc, **parsec**, JLex, CUP, SableCC,…

Why automation?