Lexical Elements & Operators

C Compiler

- Syntax of the language
  - Rules for putting words and punctuation to make correct statements

- Compiler
  - A program that checks on the legality of C code
  - If errors, compiler prints error messages and stops
  - If NO errors, compiler translates the C code into object code

C Program

- A sequence of characters that will be converted by C compiler to object code
- Compilers first collects the characters of the C program into tokens
- 6 kinds of tokens
  - Keywords
  - Identifiers
  - Constants
  - String constants
  - Operators
  - Punctuators

Characters used in a C Prog.

- Lowercase letters
- Uppercase letters
- Digits
- Other characters

  + - * / = ( ) { } [ ] < > ‘ “’

  ! # % & _ | ^ ~ \ . , ; : ?
- White space characters

  blank, newline, tab, etc.

Comments

- Arbitrary strings of symbols placed btwn /* and */

```
/*  comment */              /** another comment ***/
/******************************
 * If you wish, you can        *
 * put commas in a box.        *
 ******************************/
```

- The complier changes each comment into a single blank character
- Used by programmer as a documentation aid for explaining clearly
  - how the program works
  - how it is to be used

Keywords

- Reserved words
  - have a strict meaning as individual tokens in C
  - cannot be redefined or used in other contexts

Keywords				
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Identifiers (1/2)

- A token is composed of a sequence of letters, digits, and the special character _ (*underscore*)
- A letter or underscore must be the first character of an identifier
- Lowercase and uppercase are distinct

< Examples >	< NOT Examples>
k	not#m2
_id	101_south
iamanidentifier2	-plus
so_am_i	

Identifiers (2/2)

- Give unique names to objects in a program.
- Keywords can be thought of as identifiers that are reserved to have special meaning
  - e.g.) **printf**
- The identifier **main** is special.
- Choose names that are meaningful!!

  tax = price * tax_rate

- Identifier beginning with an underscore
  - Usually used for system names. (e.g. _iob)
  - Please do NOT begin with an undescore!

Constants (1/2)

- Integer constants

  0, **17**

- Floating constants

  1.0, **3.14159**

- Character constants
  - Written between single quotes
    - **‘a’**, **‘b’**, **‘c’**
    - closely related to integers
  - Special character constants
    - **\n** (newline)
    - Backslash is the escape character (“escaping the usual meaning of n”)

Constants (2/2)

- Integer constants
  - Decimal integers 0, 17
  - Octal integers 017
  - Hexadecimal integers 0x17

  - How about -49 ? Constant expression

String Constants

- A sequence of characters enclosed in a pair of double-quote marks
  - **“abc”**
  - collected as a single token
  - **‘a’** and **“a”** are NOT the same.

<Examples >

```
“a string of text”
“”
“        ”
“/* this is not a comment */”
“a string with double quotes \” within”
“a single backslash \\ is in this string”
```

<wrong Examples>

```
/*”this is not a string”*/
  “and
neither is this”
```

Operators & Punctuators (1/2)

- Arithmetic Operators

  + , - , * , / ,%

  (e.g.) 5%3 has the value 2.

- Operators can be used to separate identifiers

  a+b (or, **a + b**) /*an expression*/

  a_b /* a 3-character identifier*/

- Some symbols have meanings that depend on context

  printf("%d", a);

  a = b % 7;

Operators & Punctuators (2/2)

- Punctuators
  - parentheses, braces, commas, and semicolons
- Operators and punctuators, along with white space, serve to separate language elements

```
int main(void)
{
    int a, b = 2, c = 3;
    a = 17 * (b + c);
    ……
```

✓The parentheses following main are treated as an operator.

✓The symbols "{", "}", ",", ";", "(", ")" are punctuators

- Some special char.s are used in many different contexts

```
a + b      ++a      a += b
```

Precedence and Associativity of Operators

- Precedence: 연산의 우선순위
- Associativity: 연산의 방향

- Parentheses can be used to clarify or change the order in which operators are performed.

  1 + 2 * 3 ⇔ 1 + (2 * 3)

  (1 + 2) * 3

  1 + 2 – 3 +4 – 5 ⇔ (((1+2) – 3) + 4) -5

- Binary operators + and – have the same precedence, the associativity rule “left to right” is used.

Precedence and Associativity of Operators

Operator precedence and associativity	
Operator	Associativity
() ++ *(postfix)* -- *(postfix)*	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
= += -= *= /= etc.	right to left

- a * b – c unary minus sign, binary subtraction

((- a) * b) – c

Increment and Decrement Operators (1/3)

- ++ and –- are unary operators, and can be applied to variables but not to constants or expressions

<Examples>	**<wrong Examples>**
++i	**777++**
cnt--	**++(a * b -1)**

Increment and Decrement Operators (2/3)

- Difference btwn **++i** and **i++**
  - The expression **++i** causes the stored value of **i** to be incremented first, then taking as its value the new stored value of **i.**
  - The expression **i++** has as its value the current value of **i**; then the expression causes the stored value of **i** to be incremented.

```
int        a, b, c = 0;
a = ++c;
b = c++;
printf("%d %d %d\n", a, b, ++c);  /* 1 1 3 is printed */
```

Increment and Decrement Operators (3/3)

- **++** and –- cause the value of a variable in memory to be changed (<u>side effect</u>)
- Other operators do NOT do this (Ex. **a + b)**
- All three statements are equivalent.
  - ++i;   i++;   i = i + 1;

Declarations and Initializations		
int a = 1, b=2, c=3, d=4;		
Expression	Equivalent expression	Value
a * b / c	(a * b) / c	
a * b % c +1	(((a * b) % c) +1	
++a * b - c--	((++a) * b) -(c--)	
7 - -b * ++d	7 - ((-b) * (++d))	

Assignment Operators (1/2)

- Assignment expression: *variable* = *right_side*
  - = is treated as an operator
  - *right_side* is itself expression
  - The value of *right_side* is assigned to *variable*

```
b = 2;
c = 3;          ⇔     a = (b = 2) + (c = 3);
a = b + c;
```

  - “right to left” associativity

```
a = b = c = 0;   ⇔   a = (b = (c = 0));
```

Assignment Operators (2/2)

<table>
<tr><th>Assignment operators</th></tr>
<tr><td>=   +=   -=   *=   /=   %=   >>=   <<=   &=   ^= |=</td></tr>
</table>

variable op= expression ⇔ *variable* = *variable op* (*expression*)

```
j *= k + 3;    ⇔   j = j * (k+3);          /* NOT j = j * k+3; */

int i =1, j = 2, k = 3, m = 4;
i += j + k;        ⇔  i += (j + k);          ⇔   i = i + (j + k);    /* 6 */
j *= k = m + 5;  ⇔  j *= (k = (m + 5));  ⇔  j = j * (k = (m + 5));  /*18*/
```