ta1 -1

Thursday, January 20, 2022 10:53 AM

Senami Hodonu
TA-1

Edition 4: question 1.1, page 38.

Give examples of various kinds of errors. Use Java for your examples.

Question 1.1, page 38.
Give examples of various kinds of errors. Use Java for your examples.

Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.

a. A lexical error, detected by the scanner
b. A syntax error, detected by the parser
c. A static semantic error, detected by semantic analysis
d. A dynamic semantic error, detected by code generated by the compiler
e. An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)

a.

```
/**
 *A lexical error, detected by the scanner
 */
public class ErrorChecking{
        public static void main(String[] args) {
                int 'number = 7; //Lexical error due to the invalid character, " ' " .

        }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Invalid character constant

        at ErrorChecking.main(ErrorChecking.java:7)
```

b.

```
/**
 * A syntax error, detected by the parser
 */
public class ErrorChecking {
        public static void main(String[] args) {
                int 1number = 7; //syntax error due to the invalid variable name.
        }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Syntax error on token "1", delete this token

        at ErrorChecking.main(ErrorChecking.java:7)
```

c.

```
/**
 * A static semantic error, detected by semantic analysis
 */
public class ErrorChecking {
	public static void main(String[] args) {
		int number;
		number++; //variable is not initialized
	}
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The local variable number may not have been initialized

	at ErrorChecking.main(ErrorChecking.java:8)
```

d.

```
/**
 * A dynamic semantic error, detected by code generated by
 * the compiler
 */
public class ErrorChecking {
	public static void main(String[] args) {
		int[] number = new int[10];
		number[10] = 7; //Array out of bounds
	}
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 10
	at ErrorChecking.main(ErrorChecking.java:8)
```

e.

```
/**
 * An error that the compiler can neither catch nor easily
 * generate code to catch (this should be a violation of
 * the language definition, not just a program bug)
 */
public class ErrorChecking {

	public static int addVariables(int a, int b, int c) {
		return a+b+c;
	}

	public static void main(String[] args) {
		addVariables(1,2,3);
		System.out.println(addVariables);
	}
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	addVariables cannot be resolved to a variable

	at ErrorChecking.main(ErrorChecking.java:13)
```

Edition 4: question 1.8, page 39.

Consider the accuracy of `make`.

Note that plain-old `make` relies on operating-system timestamps.

The Unix make utility allows the programmer to specify dependences among the separately compiled pieces of a program. If file A depends on file B and file B is modified, make deduces that A must be recompiled, in case any of the changes to B would affect the code produced for A. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?

How accurate is this sort of dependence management?

Make provides a very helpful utility tool for managing and maintaining computer programs. As the number of files in a program increases, so to does the compile time, complexity of compilation command and the likelihood of human error. Make allows a user to automatically rebuild program (a file and all it dependencies) whenever one of the program's component files is modified. Modification in one file will effect a change in all the dependency files when compiled. This allows all the to files become accurate and up to date. Because if one is modified another file is also modified and is recompiled.

Under what circumstances will it lead to unnecessary work?

The general rule for the Make utility is to re-compile a modified file and all it dependency files. When there's a minor change in a file that effects little to no changes to its dependencies, then recompilation of its dependencies is essentially unnecessary work.

Under what circumstances will it fail to recompile something that needs to be recompiled?

Make could fail when it is run with the "--ignore error" flag. Ignoring errors make result in Make treating an error return just like success return. When an error happens that make has not been told to ignore, it implies that the current target cannot be correctly remade, and neither can any other that depends on it either directly or indirectly. No further Make recipes will be executed for these targets, since their preconditions have not been achieved. Also, in cases where the interdependencies between files are not properly specified, the recompilation of files may fail.

Edition 4: question 2.1 (a,b,c, page 105.

Give regular expressions for character-string literals in C, comments in Pascal, and numeric literals in C.

For each of the three parts, start by giving some example strings in the language.

2.1. Write regular expressions to capture the following.

a. Strings in C. These are delimited by double quotes ("), and may not contain newline characters. They may contain double-quote or backslash characters if and only if those characters are “escaped” by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is not a member of a small specified set.

$Examples := "Happy\ days!","I","his//hers"$
$Strings\ in\ C := "(Character \mid Allowable)*"$
$NonMemberCharacter := newline\ character\ (\backslash n),"\ or\ \backslash$
$Allowable :=\ \backslash" \mid \backslash\backslash$

$\boldsymbol{Character := All\ character\ except\ NonMemberCharacters}$

b. Comments in Pascal. These are delimited by (* and *) or by { and }. They are not permitted to nest.

Examples
(* *This is a comment example in pascal.* *)

{*This also a comment example in pascal* }

$NotPermitted \coloneqq \{Not(**)\} \mid (*\ Not\{\}\ *)$

Comments in Pascal $\coloneqq \{\} \mid (**)$

c. Numeric constants in C. These are octal, decimal, or hexadecimal integers, or decimal or hexadecimal floating-point values. An octal integer begins with 0, and may contain only the digits 0–7. A hexadecimal integer begins with 0x or 0X, and may contain the digits 0–9 and a/A– f/F. A decimal floating-point value has a fractional portion (beginning with a dot) or an exponent (beginning with E or e). Unlike a decimal integer, it is allowed to start with 0. A hexadecimal floating-point value has an optional fractional portion and a mandatory exponent (beginning with P or p). In either decimal or hexadecimal, there may be digits to the left of the dot, the right of the dot, or both, and the exponent itself is given in decimal, with an optional leading + or - sign. An integer may end with an optional U or u (indicating "unsigned"), and/or L orl (indicating "long") or LL or ll (indicating "long long"). A floating point value may end with an optional F or f (indicating "float"—single precision) or L or l (indicating "long"—double precision).

Examples:

$decimal\ constants \coloneqq 43\ , 48, 100$
$octal\ constants \coloneqq 033, 0188, 0144$
$hexadecimal\ constans \coloneqq 0x25, 0X37a, 0xFF, 0xAB$
$decimal\ floating\ value \coloneqq 3.14, 899.0, -0.895$
$floating\ point\ constants\ in\ Exponential\ form \coloneqq 100e6, -0.76E27, -8.55e18$
$unsigned\ integer: =\ 23u, 034U, 0x8u$

integer_values = (oct_integers | deci_integers | hexa_integers) int_suffix

oct_integer = 0 oct_digits*

oct_digits = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

deci_integer = (1 |2 | 3 |4 | 5 |6 | 7 |8 | 9) deci_digits*

deci_digits = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

hexa_integer = (0x or 0X) hexa_digit*

hexa_digit = deci_digits | a | b | c | d | e | f | A | B | C | D | E | F

int_suffix $\coloneqq \epsilon$ | ((U | u) | (L | l | LL | ll | ϵ)) | ((U | u) (L | l | LL | ll | ϵ))

floating-point_values := (dec_fpv | hexa_fpv) fp_suffix

dec_(fpv)$\coloneqq$ deci_digit* (.deci_digit |deci_digit.) deci_digit* ((E|e)exponent | ϵ) | dec_digit dec_digit*(E |e) exponent

hexa_fpv ≔ (0x|0X) hexa_digit∗ (.hexa_digit | hexa_digit. | ϵ) hexa_digit ∗ (P|p) exponent

exponent = (optional_sign) deci_digit deci_degit∗

optional_sign≔ + | - | ϵ

fp_suffix = ϵ | F | f | L | l

Numeric constants in C = interger_values | floating-point_values

Edition 4: question 2.13 (a,b), page 108.

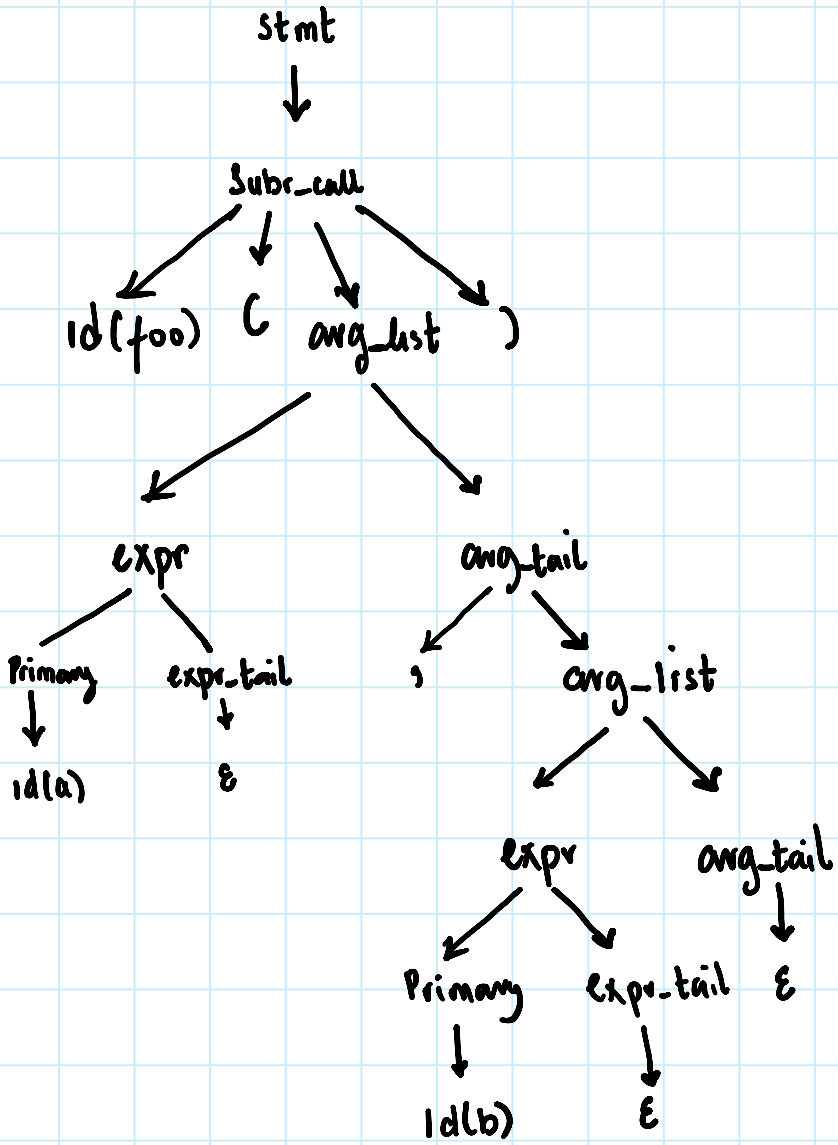
Give a parse tree and a rightmost derivation, of a string, according to a context-free grammar.

Note that we do not cover grammar characteristics, like LL(1).

Consider the following grammar:

stmt ⟶ *assignment*
⟶ *subr_call*
assignment ⟶ `id := expr`
subr_call ⟶ `id ( arg_list )`
expr ⟶ *primary expr_tail*
expr_tail ⟶ *op expr*
⟶ ϵ
primary ⟶ `id`
⟶ *subr_call*
⟶ `( expr )`
op ⟶ `+ | - | * | /`
arg_list ⟶ *expr args_tail*
args_tail ⟶ `, arg_list`
⟶ ϵ

a. Construct a parse tree for the input string foo(a, b).



b. Give a canonical (right-most) derivation of this same string.

Stmt ⇒ Sub_call
⇒ id(arg_list)
⇒ id(expr arg_tail)
⇒ id(expr , arg_list)
⇒ id(expr, expr arg_tail)
⇒ id (expr, primary expr_tail ε)

=> id(expr, id ε)

=> id(primary expr_tail, id)

=> id(id ε, id)

=> id(id, id)

=> foo(a, b)

Edition 4: question 2.17, page 109.

Extend the context-free grammar from Figure 2.25, which is on page 91, which is in Section 2.3. Note that we do not cover Section 2.3, but we can still use the grammar.

Extend the grammar of Figure 2.25 to include if statements and while loops, along the lines suggested by the following examples:

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an if or while statement.

1. *program* ⟶ *stmt_list* $$
2. *stmt_list* ⟶ *stmt_list stmt*
3. *stmt_list* ⟶ *stmt*
4. *stmt* ⟶ `id :=` *expr*
5. *stmt* ⟶ `read id`
6. *stmt* ⟶ `write` *expr*
7. *expr* ⟶ *term*
8. *expr* ⟶ *expr add_op term*
9. *term* ⟶ *factor*
10. *term* ⟶ *term mult_op factor*
11. *factor* ⟶ `(` *expr* `)`
12. *factor* ⟶ `id`
13. *factor* ⟶ `number`
14. *add_op* ⟶ `+`
15. *add_op* ⟶ `-`
16. *mult_op* ⟶ `*`
17. *mult_op* ⟶ `/`

1. stmt → if condition then fi
2. condition → expr relation expr
3. *expr → term*
4. *expr → number*
5. stmt_list → expr add_op term
6. expr → term
7. term → number
8. stmt → while condition do stmt_list fi
9. *condition → expr relation expr*
10. *expr → term*
11. *expr → number*
12. stmt_list → *stmt*
13. stmt → *read id*
14. stmt → expr add_op term
15. stmt → expr add_op term
16. expr → *term*
17. term → *number*
18. *relation* → < | > | <= |>=| = | ! =