

Chapter 3:

C Syntax I: Expressions, Statements

References:

- Kernighan/Ritchie: The C Programming Language, 2nd Ed. [1988]
Chapter 2: Types, Operators, and Expressions (pages 35–54)
Appendix A7: Reference Manual: Expressions (pages 199–210)

Overview

1. Operator Priority and Associativity

2. Context-Free Grammars

3. Statements in C

Expressions

- Expressions are syntactic constructs that describe values. Expressions consist of:
 - ◇ Constants, e.g. `-123`, `'a'`, `"abc"`.
 - ◇ Variables, e.g. `i`, `a_variable`, `x2`.
 - ◇ Operators, e.g. `+`, `*`.

In C, also `=` is an operator. In many other languages, `=` is not an operator, only part of an assignment statement.
 - ◇ Function calls, e.g. `strlen(s)`.
 - ◇ Parentheses: `(` and `)`.

Operator Type (1)

- An **operator** is the name of a function, e.g. $+$ denotes the addition function $f(x, y) = x + y$.
- The **operands** of an operator are its input values (function arguments or parameters).
- Operators use a special syntax for the function call:
 - ◇ Functions are normally called in the form $f(x)$, $f(x_1, x_2)$, and so on, e.g. `strcpy(buf, "xyz")`.
 - ◇ However, operators can be placed before, in between, or after their input values.

The exact form depends on the operator type.

Operator Type (2)

- Most programming languages use operators for a small set of elementary, built-in functions.
- The standard function-call syntax $f(x_1, \dots, x_n)$ is utilized for user-defined or library functions.
- Operators can take different numbers of operands:
 - ◇ **Monadic (unary)** operators have one operand.
 - ◇ **Binary** operators have two operands.
 - ◇ Operators of higher **arity** are seldom.

Arity is the number of operands. E.g. the conditional expression in C “`_?_:_`” can be seen as operator of arity 3.

Operator Type (3)

- Depending on the position of the operator relative to its operands, operator types are defined:
 - ◇ **Infix** operators are binary and are written between their operands, e.g. $x + 1$.
 - ◇ **Prefix** operators are monadic and written before their operand, e.g. $*x$, $\&x$, $-x$, $++x$.
 - ◇ **Postfix** operators are monadic and written after their operand, e.g. $x++$ and (not in C): $5! = 120$.
 - ◇ **Mixfix** operators consist of several parts.
E.g. $a[i]$ is the operator $[\]$ applied to a , i .

Operator Type (4)

- Operators usually consist of special characters.

In contrast, standard function names must be identifiers (consisting of letters and digits).

- C has a fixed set of operators.
- C++ also has a fixed set of operators, but the user can overload them with new argument types.
- In Prolog, the user can declare new operators.

In Prolog, one can always use also the standard syntax, e.g. one can write `+(2,3)`. But if `+` is declared as infix operator (it is usually already predeclared), one can use the more convenient syntax `2+3`.

Operator Priority

- The expression $5+3*2$ means $5+(3*2)$, not $(5+3)*2$.
- The reason is that $*$ has a higher priority than $+$.
Instead of priority, one can also say binding strength or precedence.
- Operators select operands in the order of priority:
 - ◇ $*$ first takes the operands that are immediately left and right of it, and protects the entire group by parentheses: $(3*2)$.
 - ◇ Then $+$ gets its chance, but because of the parentheses, it must take the entire group $(3*2)$ as right operand. The result is: $(5+(3*2))$.

Operator Associativity

- Operators of the same priority get their turn to select operands from left to right (**left associative**) or from right to left (**right associative**).

There are also **non associative** operators (require parentheses).

- E.g. + and - are usually left associative.

Therefore, $5-3-2$ means $(5-3)-2$, not $5-(3-2)$.

One should put explicit parentheses in this example (better style).

The subtraction is not associative, therefore parentheses matter.

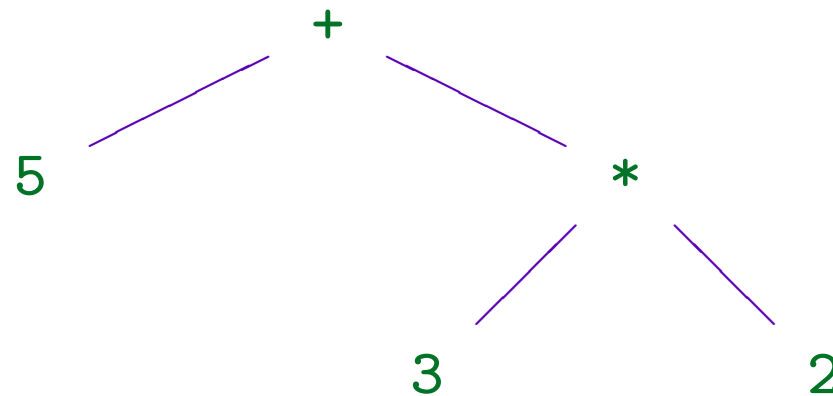
- To have a defined operand selection sequence, operators of equal priority need equal associativity.

C Operators

Name	Priority	Assoc.
() [] -> .	15 (highest)	left
! ~ ++ -- + - * & (T) sizeof (unary)	14	right
* / % (binary)	13	left
+ - (binary)	12	left
<< >>	11	left
< <= > >=	10	left
== !=	9	left
&	8	left
^	7	left
	6	left
&&	5	left
	4	left
?:	3	right
= += -= *= /= %= &= ^= = <<= >>=	2	right
,	1 (lowest)	left

Operator Trees (1)

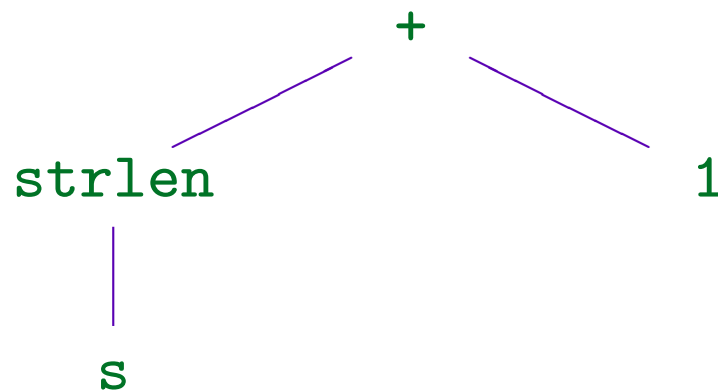
- The structure of an expression can be shown as an operator tree:



- The values flow from the leaves (at the bottom) towards the root (at the top). E.g. the inputs of $*$ are 3 and 2, the inputs of $+$ are 5 and 6 (i.e. $3*2$).

Operator Trees (2)

- Of course, operator trees can contain variables and function calls:



- A value is computed at each node (bottom up).
 - If the node is a variable, the current value of the variable is taken and passed to the parent node.
- The value at the root is the value of the expression.

Side Effects

- The computation of a value is usually considered the main purpose of an expression.
- However, an expression can have “side effects”, i.e. change the values of variables or do input/output.

Some languages exclude this, e.g. forbid that functions that can be called in expressions may have any side effects. However, in C side effects are the usual case, since even assignments are permitted inside expressions.

- E.g. “`i++`” returns the current value of `i`, but has the side effect of incrementing `i` (`$i = i + 1$`).

Exercise

Draw operator trees for the following C expressions:

- `c = *p++`
- `1900 <= year && year < 2000`
- `(c = getc(stdin)) != EOF`

Overview

1. Operator Priority and Associativity

2. Context-Free Grammars

3. Statements in C

Context-Free Grammars (1)

- A context-free grammar (CFG) is a quadruple $G = (N, T, P, S)$ where
 - ◇ N is an alphabet,
 N is called the set of nonterminals or nonterminal symbols.
 - ◇ T is an alphabet with $N \cap T = \emptyset$,
 T is called the set of terminals or terminal symbols.
 - ◇ $P \subseteq N \times (N \cup T)^*$,
 P is called the set of productions or syntax rules. Productions are usually written in the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.
 - ◇ $S \in N$.
 S is called the start symbol.

Context-Free Grammars (2)

- Given a CFG G , a relation \Rightarrow is defined on $(N \cup T)^*$ by $\alpha \Rightarrow \beta$ if and only if there are $\alpha_1, \alpha_2 \in (N \cup T)^*$ and $A \rightarrow \gamma$ in P such that

$$\begin{aligned}\alpha &= \alpha_1 A \alpha_2, \\ \beta &= \alpha_1 \gamma \alpha_2.\end{aligned}$$

- \Rightarrow^* is the reflexive and transitive closure of \Rightarrow .

I.e. $\alpha \Rightarrow \beta$ if and only if there are $n \in \mathbb{N}_0$ and $\gamma_0, \dots, \gamma_n \in (N \cup T)^*$ such that $\gamma_0 = \alpha$, $\gamma_n = \beta$ and $\gamma_{i-1} \Rightarrow \gamma_i$ for $i = 1, \dots, n$.

- $L(G) = \{\alpha \in T^* \mid S \Rightarrow^* \alpha\}$ (language defined by G).

The sequence $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = \alpha$ is called a derivation of α .

Context-Free Grammars (3)

- E.g. a very simple grammar for expressions is $G = (N, T, P, S)$ where
 - ◇ $N = \{E\}$ (of course, E is also the start symbol S),
 - ◇ $T = \{\text{id}, \text{iconst}, +, -, *, /, \%, (,)\}$.

id and iconst (as all tokens) are terminal symbols for the CFG, but they are actually replaced by a character sequence defined by a regular expression (outside the CFG).

- ◇ P consists of the following productions:

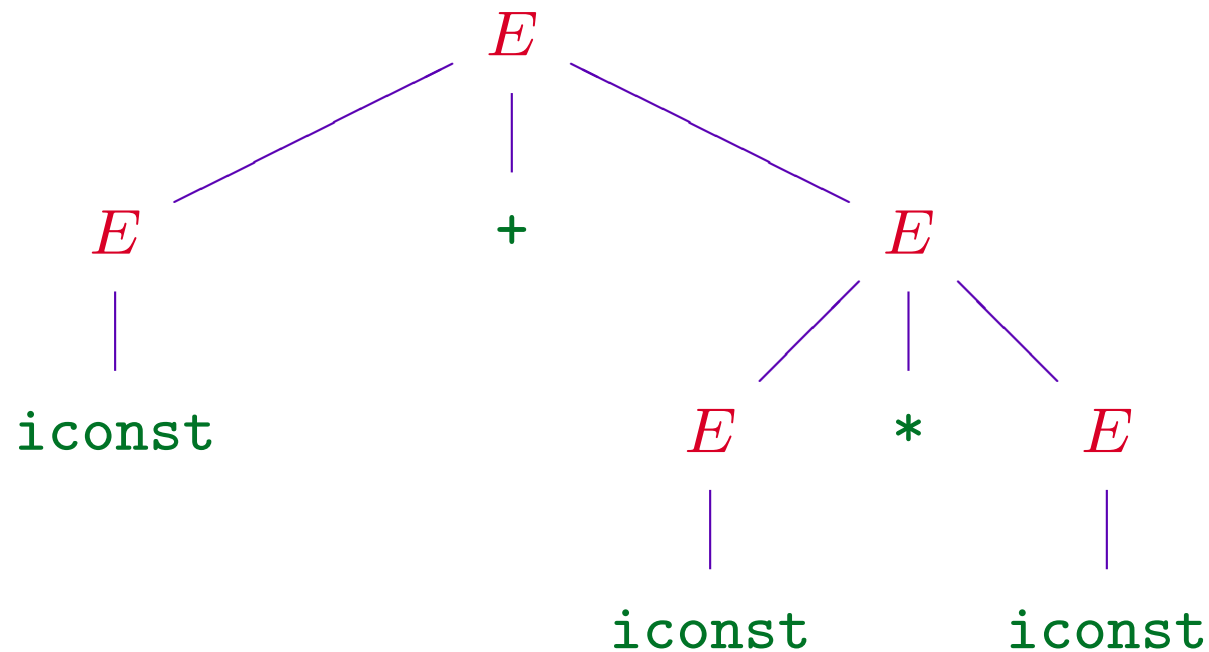
$$\begin{array}{ll} E \rightarrow \text{id} & E \rightarrow E * E \\ E \rightarrow \text{iconst} & E \rightarrow E / E \\ E \rightarrow E + E & E \rightarrow E \% E \\ E \rightarrow E - E & E \rightarrow (E) \end{array}$$

Derivation Trees (1)

- A tree is a **derivation tree** of $\alpha \in T^*$ (wrt G) iff
 - ◇ Each inner node is labelled with a symbol from N , each leaf node with an element of $T \cup \{\epsilon\}$.
 - ◇ The root node is labelled with the start symbol S .
 - ◇ For each inner node X the following holds: Let X be labelled with A and let the child nodes of X be labelled with $\alpha_1, \dots, \alpha_n$ (from left to right). Then $A \rightarrow \alpha_1 \dots \alpha_n$ is a production of G .
 - ◇ The labels of the leaf nodes read from left to right (with ϵ nodes removed) form the word α .

Derivation Trees (2)

- Example: Derivation tree (or “parse tree”) for “`iconst + iconst * iconst`”, e.g. `5+3*2`:



Derivation Trees (3)

- A derivation tree contains the important information in a derivation: Which production was applied to which (occurrence of a) nonterminal symbol.

It does not contain the sequence in which nonterminals were replaced.

- E.g. the tree on the previous slide corresponds to both of the following derivations (and more):

$$\diamond E \Rightarrow E + E \Rightarrow \text{iconst} + E \Rightarrow \text{iconst} + E * E \Rightarrow \text{iconst} + \text{iconst} * E \Rightarrow \text{iconst} + \text{iconst} * \text{iconst}.$$

$$\diamond E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \text{iconst} \Rightarrow E + \text{iconst} * \text{iconst} \Rightarrow \text{iconst} + \text{iconst} * \text{iconst}.$$

Derivation Trees (4)

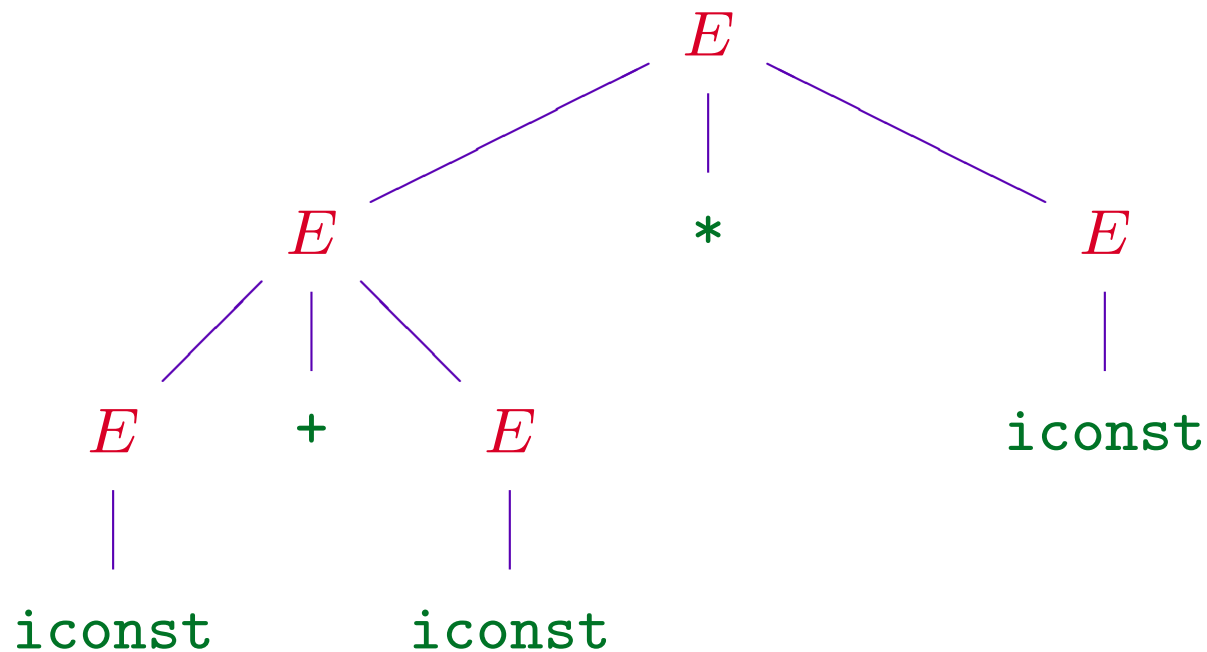
- A **leftmost derivation** is a derivation where in each step the first nonterminal is replaced.
- A **rightmost derivation** is a derivation where in each step the last nonterminal is replaced.
- There is a unique leftmost (rightmost) derivation for every derivation tree.

By permitting only leftmost (or only rightmost) derivations, the non-essential information about the replacement sequence is removed.

- In any case, there is a unique derivation tree for every derivation.

Ambiguous Grammars (1)

- “`iconst + iconst * iconst`” has also the following derivation tree with respect to the above grammar:



Ambiguous Grammars (2)

- A grammar G is called **ambiguous** if and only if there is a word $\alpha \in L(G)$ such that there are at least two derivation trees for α .
- The above grammar for expressions is ambiguous.
- This is bad, because **the structure of the derivation tree can be used in the compiler.**
- E.g. the first derivation tree above is very similar to the correct operator tree for the expression.

The second derivation does not respect that $*$ takes precedence over $+$.

Ambiguous Grammars (3)

- The purpose of the **syntax analysis phase** of the compiler (“**parser**”) is not only to check whether the input program is syntactically correct.
- It also **constructs a derivation tree** that is used by the semantic analysis and code generation.

Since the compiler phases run today interleaved, it might be that the derivation tree never exists as a whole, because constructed parts are immediately processed.

- There can be different grammars for the same language: G_1 and G_2 are equivalent iff $L(G_1) = L(G_2)$.

Ambiguous Grammars (4)

- One usually tries to find an unambiguous grammar for a programming language.

Otherwise the output of the syntax analysis would not be well-defined. However, in practice parser-generator tools like yacc have disambiguation rules in order to select a specific derivation tree for every word. But then the compiler writer must know these rules exactly, and carefully study every warning about “conflicts” in the parser construction.

- In addition, the derivation tree should describe a “natural” structure of a program.

I.e. a structure that is useful for the compiler. For expressions this means a structure that is very similar to the operator tree. The operator tree can be used for type checking and code generation.

Grammar for Expressions (1)

- The following grammar for expressions is **unambiguous** and **respects operator priorities**:

◇ $N = \{E, T, F\}$ where E is the start symbol.

E stands for “expression”, T for “term”, and F for “factor”.

◇ $T = \{\text{id}, \text{iconst}, +, -, *, /, \%, (,)\}$.

◇ P consists of the following productions:

$$E \rightarrow E + T$$

$$T \rightarrow T \% F$$

$$E \rightarrow E - T$$

$$T \rightarrow F$$

$$E \rightarrow T$$

$$F \rightarrow \text{iconst}$$

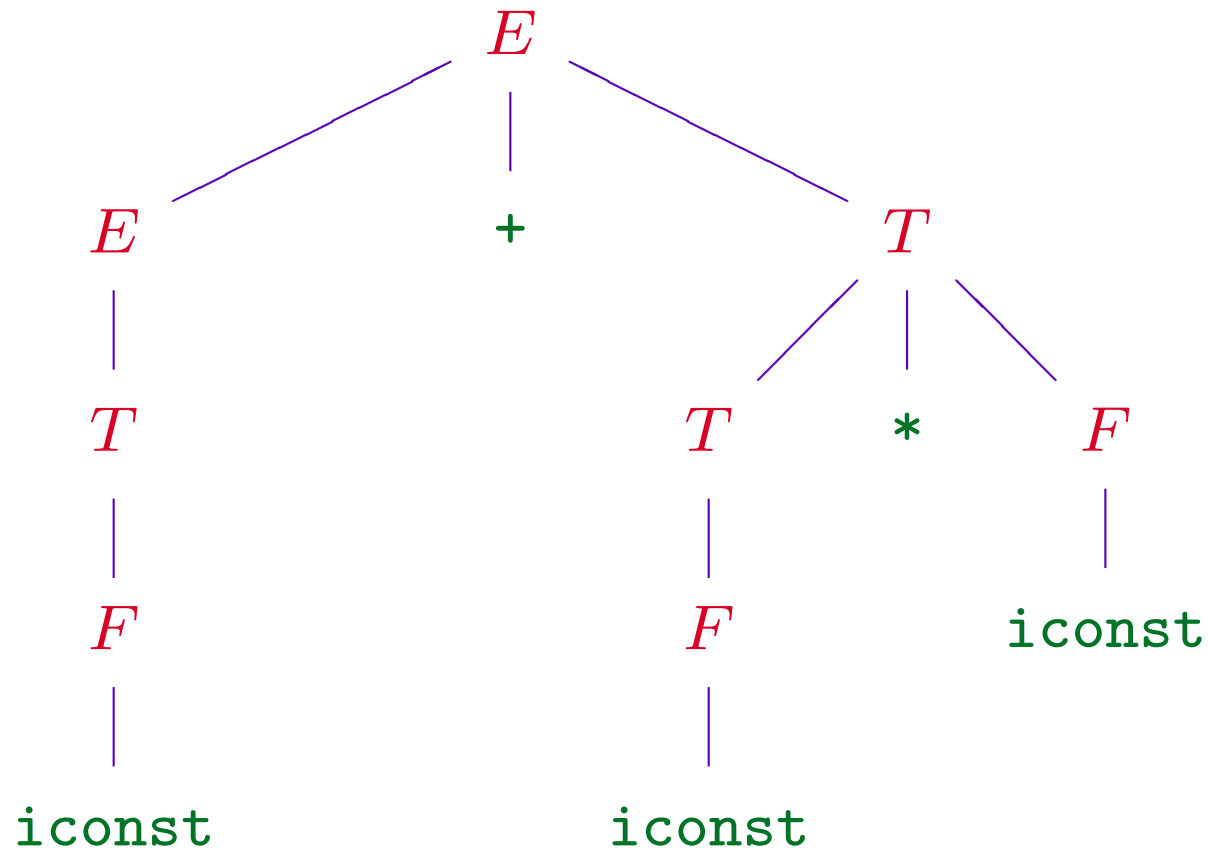
$$T \rightarrow T * F$$

$$F \rightarrow \text{id}$$

$$T \rightarrow T / F$$

$$F \rightarrow (E)$$

Grammar for Expressions (2)



Grammar for Expressions (3)

- The trick is to replace the single nonterminal E for expressions by several: One for each priority level plus one for the atomic expressions.
- In the example grammar,
 - ◇ E corresponds to the lowest priority level 1 and is responsible for generating $+$ and $-$.
 - ◇ T : priority level 2, generates $*$, $/$, and $\%$.
 - ◇ F corresponds to the highest priority level 3 and constructs the atomic expressions `iconst` and `id` as well as explicitly parenthesized expressions.

Grammar for Expressions (4)

- With the productions $E \rightarrow T$ and $T \rightarrow F$, one can switch freely to symbols of higher priority, but the only way back is through parentheses: $F \rightarrow (E)$.
- This excludes the derivation tree with “*” above “+”: Once one has generated “*”, the left and right side are T and F and it is no longer possible to generate “+” below (except inside “(” and “)”).
- Because T is on the left side, “*” is left associative: The left subtree can contain another “*”, but the right subtree cannot.

Expressions in C (1)

- Terminal symbols (tokens) are marked with boxes.

- expression \rightarrow expression , assignment

expression \rightarrow assignment

A, B means to execute first A, then B. It is very similar to A; B;, but that would be two statements whereas A, B remains an expression.

- assignment \rightarrow unary-expr assign-op assignment

assignment \rightarrow cond-expr

- assign-op: One of

=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=.

Expressions in C (2)

- `cond-expr` \rightarrow `or-expr` `?` `expression` `:` `cond-expr`
`cond-expr` \rightarrow `or-expr`

`A ? B : C` means if A is true, then B, else C. In contrast to the similar if-statement, it is an expression and has a value (B or C).

- `or-expr` \rightarrow `or-expr` `||` `and-expr`
`or-expr` \rightarrow `and-expr`
- `and-expr` \rightarrow `and-expr` `&&` `bitor-expr`
`and-expr` \rightarrow `bitor-expr`

Expressions in C (3)

- `bitor-expr` → `bitor-expr` `|` `bitxor-expr`
`bitor-expr` → `bitxor-expr`
- `bitxor-expr` → `bitxor-expr` `^` `bitand-expr`
`bitor-expr` → `bitand-expr`
- `bitand-expr` → `bitand-expr` `&` `comparison`
`bitand-expr` → `comparison`

Expressions in C (4)

- comparison → comparison `==` unequality
- comparison → comparison `!=` unequality
- comparison → unequality
- unequality → unequality `uneq-op` shift-expr
- unequality → shift-expr
- `uneq-op`: One of `<`, `<=`, `>`, `>=`.

Expressions in C (5)

- `shift-expr` \rightarrow `shift-expr` `<<` `add-expr`
- `shift-expr` \rightarrow `shift-expr` `>>` `add-expr`
- `shift-expr` \rightarrow `add-expr`
- `add-expr` \rightarrow `add-expr` `+` `mult-expr`
- `add-expr` \rightarrow `add-expr` `-` `mult-expr`
- `add-expr` \rightarrow `mult-expr`

Expressions in C (6)

- mult-expr \rightarrow mult-expr * cast-expr
 mult-expr \rightarrow mult-expr / cast-expr
 mult-expr \rightarrow mult-expr % cast-expr
 mult-expr \rightarrow cast-expr
- cast-expr \rightarrow (type-name) unary-expr
 cast-expr \rightarrow unary-expr

Expressions in C (7)

- unary-expr → ++ unary-expr
- unary-expr → -- unary-expr
- unary-expr → unary-op cast-expr
- unary-expr → sizeof unary-expr
- unary-expr → sizeof (type-name)
- unary-expr → postfix-expr
- unary-op: One of &, *, +, -, ~, !.

Expressions in C (8)

- postfix-expr → postfix-expr ++
- postfix-expr → postfix-expr --
- postfix-expr → postfix-expr [expression]
- postfix-expr → postfix-expr (arguments)
- postfix-expr → postfix-expr . identifier
- postfix-expr → postfix-expr -> identifier
- postfix-expr → primary-expr

Expressions in C (9)

- arguments \rightarrow arg-list

arguments $\rightarrow \epsilon$

If a function is called in C that has 0 arguments (parameters), one nevertheless must put “()” after the function name to clearly mark that this is a function call.

- arg-list \rightarrow arg-list , assignment

arguments \rightarrow assignment

“expression” as argument would permit the comma operator, and then the grammar would become ambiguous. Therefore “assignment” is the maximal expression that can be permitted without parentheses.

Expressions in C (10)

- `primary-expr` → `identifier`
- `primary-expr` → `integer-constant`
- `primary-expr` → `character-constant`
- `primary-expr` → `enum-constant`
- `primary-expr` → `float-constant`
- `primary-expr` → `string-constant`
- `primary-expr` → `(expression)`

Constant Expressions (1)

- The compiler must know the size of an array, therefore in the declaration “`char a[S];`” the size S must be a constant expression.
- In C, constant expressions are very restricted, see next slide. But one can use e.g. `+`, `-`, `sizeof`, `?:..`

C even excludes write-protected variables: “`const n = 5;`”, although some compilers accept this. In C, real constants can be defined with the precompiler (see next chapter). The precompiler replaces certain names by their definitions, so that e.g. `N` can be replaced by `5` before the compiler sees the program. One of the design goals of C++ was to remove the need for the precompiler, and there “`const n = 5;`” declares a real constant that can be used in constant expressions.

Constant Expressions (2)

- Constant expressions are restricted as follows:
 - ◇ For “primary-expr” identifiers (except enumeration constants) and string constants are excluded. If a float-constant is used, there must be a cast to integer.

This implies that also the following is not possible: Array accesses, structure accesses, the dereference operator “*”, the address operator “&”, assignments, the increment operator “++”, the decrement operator “--”, function calls. Furthermore, the comma operator is excluded, because it is only meaningful together with assignments.

- ◇ But as operand of `sizeof`, any expression is valid.

Constant Expressions (3)

- So one can **formally define constant expressions** by repeating the relevant parts of the above grammar with each nonterminal prefixed “const”.
- When certain variables are initialized (global, static, arrays, structures, see below), the initial value must also be a constant expression.
- There, the address operator “&” is permitted if applied to a previously declared external or static variable or function. Then pointer arithmetic is possible.

Overview

1. Operator Priority and Associativity

2. Context-Free Grammars

3. Statements in C

Statements in C

- Statements are instructions that usually change the state of computation (change variable values, do input/output). They do not compute a value.

Of course, expressions inside statements compute values.

- C has the following classes of statements:

statement	→	expr-stmt
statement	→	block
statement	→	cond-stmt
statement	→	loop-stmt
statement	→	labelled-stmt
statement	→	jump-stmt

Expression Statement (1)

- Languages like Pascal have an assignment statement and a procedure call statement.
- This is done in expressions in C, so the expression statement consists of an expression followed by “;”:

expr-stmt → expression ;

expr-stmt → ;

The expression can also be missing (this is called “null statement”).

Expression Statement (2)

- A compiler may issue a warning if a value is computed that is not used (e.g. assigned to a variable).
- E.g. the statement

`x + 5;`

is syntactically legal, but useless.

- In higher warning levels, one might get warnings that functions return values that are ignored.

E.g. `printf` returns the number of characters written or a negative number if an error occurred. One can do a cast to `void` to clearly declare that one is not interested in the value. However, a good program should check for all possible error conditions.

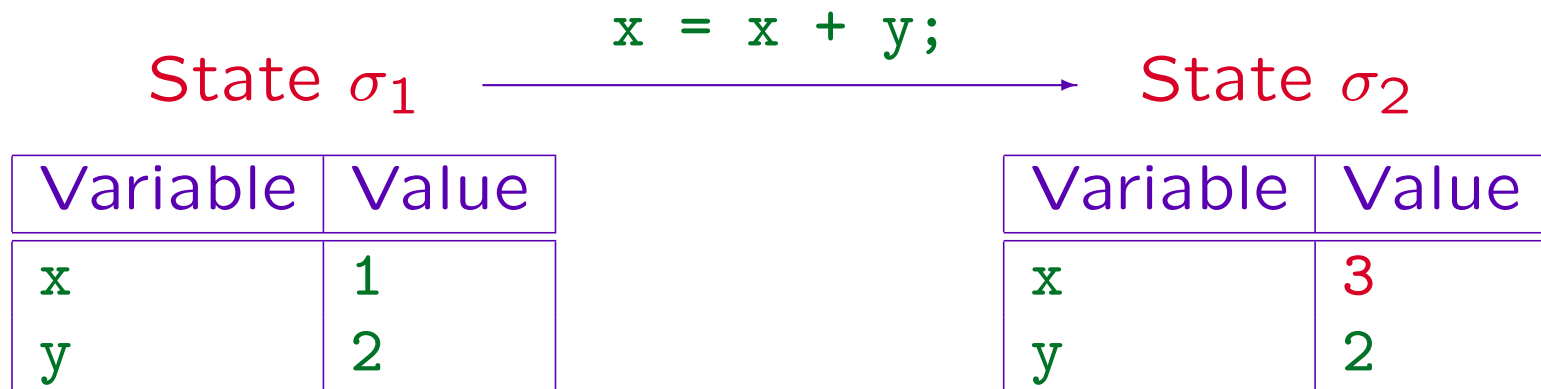
Expression Statement (3)

- As all statements, the expression statement usually changes the state of the computation.
- The **state** contains:
 - ◇ the values of all **variables**,
 - ◇ the contents of the screen (**output** so far),
 - ◇ the remaining **input** from the keyboard.

Programs that use files, attached devices like printers, network communications, etc. need a more complicated definition of state. The state is everything that the program can query or change. Definitions become more complicated if the state can also change by events that are outside to the program (e.g. when accessing shared files).

Expression Statement (4)

- E.g. the following state transformation can be done by an expression statement:



- Only the relevant part of the state is shown.

E.g. the input/output state is not important for this example.

Expression Statement (5)

- In languages like Pascal, the definition of the meaning of an assignment statement $X := E$ is simple:
 - ◇ Let the current state be σ_1 . Then the expression E is evaluated in σ_1 , let the result be v .
 - ◇ Then a new state σ_2 is computed that is identical to σ_1 except that X now has the value v .
- In C, an expression can have any number of side effects, so that actually a whole sequence of states can be required.

Evaluation Order (1)

- For most operators, C does not guarantee an evaluation sequence of the operands.
- The compiler may decide to evaluate the left operand first, but it also may decide to evaluate the right operand first (to make better use of registers).
- E.g. suppose that `n` currently has the value `5`.
Then after the statement

```
m = n++ + n;
```

the variable `m` may have the value `10` or `11`.

Evaluation Order (2)

- It is dangerous to write expressions that depend on the evaluation order: The program might first work correctly, but when it is later recompiled with a new version of the compiler, it suddenly fails.
- Even the meaning of “`a[i] = i++;`” is not clear.

Suppose that `i` is `2` before the assignment. It might be that first the right side is evaluated, and `2` is assigned to `a[3]`. But it might be that the compiler determines the address first, then `2` is assigned to `a[2]`.
- Of course, “`i = i + 1`” is well-defined: The value of a variable can only be changed after the new value is computed.

Evaluation Order (3)

- One can only rely on the following:
 - ◇ “&&” and “||” evaluate first their left operand. The right operand is only evaluated if it is still necessary to determine the result.
 - ◇ “,” first evaluates the left operand, then the right one, e.g. $X = X + 1$, $X * X$ is well-defined.
 - ◇ “A ? B : C” first evaluates A. If the result is true (i.e. not 0), B is evaluated, else C.
 - ◇ In an assignment $A = B$, the variable value is only changed after A and B are both evaluated.

Block (1)

- A **block** (or “**compound statement**”) consists of an optional sequence of declarations followed by an optional sequence of statements.
- The entire block is enclosed in “{” and “}”:

block \rightarrow $\{$ **opt-decls** **opt-stmts** $\}$

- **opt-decls** \rightarrow **decl-list**

opt-decls \rightarrow ϵ

Block (2)

- decl-list → declaration
decl-list → decl-list declaration
- opt-stmts → stmt-list
opt-stmts → ϵ
- stmt-list → statement
stmt-list → stmt-list statement

Block (3)

- In Pascal, statements are separated by “;”.

In C, all elementary statements end with a “;”.

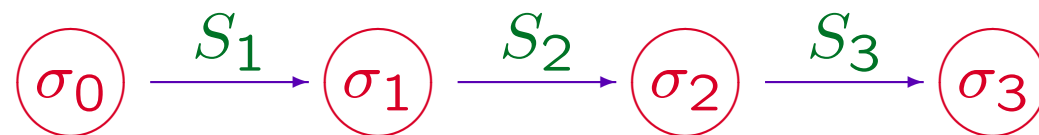
Therefore, the grammar rule for “stmt-list” above does not introduce “;”. Only the expression statement and the jump statements generate the “;” at the end. Other statements end with one of these elementary statements (and therefore with “;”) or end with a block (and therefore with “}”). Declarations also end with “;” or “}”.

- In C, all declarations in a block must be written before the first statement.

In C++, declarations and statements can be freely intermixed. Since declarations can contain initializations, this might sometimes be useful.

Block (4)

- The statements are executed in the order they are written down in the block.
- E.g. if the compound statement “{ S_1 S_2 S_3 }” starting in state σ_0 , the following state transformations are done:



- Declarations are treated later in this chapter (they may contain initializations that change the state).

If Statement (1)

- C has two types of conditional statements:
The “if” statement and the “switch” statement.
- `cond-stmt` → `if` (`expression`) `statement`
- `cond-stmt` → `if` (`expression`) `statement`
`else` `statement`
- `cond-stmt` → `switch` (`expression`) `statement`
- The “switch” statement is treated at the end of this section (needs “case” and “break”).

If Statement (2)

- The expression after `if` is called condition. It must have arithmetic type (e.g. `int`) or pointer type.

Arithmetic type includes `float`. However, equality comparisons of float values are always problematic (float values are inexact numbers).

- In C, there is no keyword “`then`”. On the other hand, C requires parentheses around the condition.
- The grammar is ambiguous for the “dangling else”:
`if(A) if(B) S1 else S2.`

The `else`-part could belong to each of the `if`. C specifies that `else` always belongs to the last open `if` (in the same block). So in the example, it belongs to “`if(B)`”. Use “`{...}`” in case of doubt.

If Statement (3)

- The statement “`if(C) S ” is executed as follows. Let the initial state be σ_0 .
 - ◇ First, the expression C (“condition”) is evaluated. This may have side effects, let the state after evaluation of C be σ_1 .
 - ◇ If the value of C is 0, σ_1 is already the final state.
 - ◇ Otherwise the statement S is executed. It transforms σ_1 into σ_2 , which is then the final state.`



If Statement (4)

- The statement “`if(C) S_1 else S_2 ” is executed as follows. Let the initial state be σ_0 .
 - ◇ First, C is evaluated. Let the state after evaluation of C be σ_1 .
 - ◇ If the value of C is not 0, the statement S_1 is executed (starting in state σ_1). The resulting state is the final state.
 - ◇ If the value of C is 0, S_2 is executed (starting in state σ_1). The resulting state is the final state.`

If Statement (5)

- The following structure is quite common:

```
if( $C_1$ )  $S_1$   
else if( $C_2$ )  $S_2$   
else if( $C_3$ )  $S_3$   
...  
else if( $C_n$ )  $S_n$   
else  $S_{n+1}$ 
```

- The conditions C_i are evaluated until one is not 0. Let C_i be the first such condition. Then S_i is executed (and none of the other statements). If all C_j are false (i.e. 0), the statement S_{n+1} is executed.

If Statement (6)

- One should indent the dependent statements below their conditions, e.g.

```
if(x > 0)
    sign = 1;
else if(x == 0)
    sign = 0;
else
    sign = -1;
```

- The compiler ignores white space and thus indentation. There will be no warning if the real structure does not correspond to the indentation.

If Statement (7)

- If several statements depend on an `if`, one must use “{” and “}” to make it a single statement (block, compound statement):

```
if(x > 0) {
    abs = x;
    negative = 0;
}
else {
    abs = -x;
    negative = 1;
}
```

If Statement (8)

- The last slide has shown the Kernighan/Ritchie style of placing “{” and “}”, which is probably followed by a majority of C programmers.
- Alternatives are these styles (need on more line):

```
if(x < 0)
{
    abs = x;
    negative = 0;
}
```

```
if(x < 0)
{
    abs = x;
    negative = 0;
}
```

- Whatever you decide, be consistent!

If Statement (9)

Exercise: Can you simplify this program part?

```
(1)  int i, j;
(2)  scanf("%d", &i);
(3)  if(i < 10)
(4)      if(i > 10)
(5)          j = 5;
(6)      else
(7)          j = 10;
(8)  else if(i <= 10)
(9)      j = i;
(10) else
(11)     j = (i & 0) + 012;
(12) printf("%d", j);
```

While Statement (1)

- C has three kinds of loop statements. Loop statements execute the dependent statement iteratively (multiple times).

• loop-stmt → while (expression) statement

loop-stmt → do statement

while (expression) ;

loop-stmt → for (opt-expr ; opt-expr ;
opt-expr) statement

While Statement (2)

- `while(C) S` is executed as follows:
 - ◇ First, the condition C is evaluated.
 - ◇ If the result is 0 (i.e. false), the loop ends.
So S was never executed.
 - ◇ If C is not 0 (i.e. C is true), S is executed.
 - ◇ Next the condition C is again evaluated. If the result is 0, the loop ends.
Then S was executed only once.
 - ◇ Otherwise the statement S is executed again.
 - ◇ And so on: S is executed as long as C is true.

While Statement (3)

- E.g. the following program prints 100 times “I should not crib (copy) my homeworks”:

```
(1)  #include <stdio.h>
(2)  int main()
(3)  {
(4)      int i = 1;
(5)      while(i <= 100) {
(6)          printf("I should not ... \n");
(7)          i++;
(8)      }
(9)      return(0);
(10) }
```

While Statement (4)

- Of course, it is important that eventually, the condition of the `while`-statement becomes false.
- Otherwise the body of the loop (the dependent statement) is executed forever (until the user kills the process or presses the reset button).

Under UNIX, one can press `Ctrl+C` (ASCII 3: ETX). Programs can choose to ignore this signal. One has a second chance with `Ctrl+\` (ASCII 28: FS). If this is switched off, too, one has to find out the process ID (with `ps`) and then call “`kill -9 ID`” (this must be done from a second window or maybe the same window after `Ctrl+Z`).

Under Windows, one can also use `Ctrl+C` or end the process with the task manager (appears on `Ctrl+Alt+Delete`).

While Statement (5)

Exercise:

- Does the following procedure make any sense?
If yes, what does it compute?

```
(1)  int f(const char *s)
(2)  {
(3)      const char *p;
(4)      p = s;
(5)      while(*p++) ;
(6)      return(p - s - 1);
(7)  }
```

While Statement (6)

- In general, it is impossible to write a compiler that warns the user if and only if a loop will not terminate.

The halting problem is a classical undecidable problem.

- Of course, a clever compiler can warn the user if the body of the loop contains no assignment to a variable that appears in the loop condition.

Then the loop condition can never change its truth value. So if it was true at the beginning, it remains true forever. However, even if loop variables change, it might be that the condition always evaluates to “true”. It is the responsibility of the programmer to make sure that all loops terminate eventually.

Do Statement (1)

- The **while**-Statement tests the condition at the beginning. Thus,
 - ◇ variables in the condition must be initialized before the loop,
 - ◇ it is possible that the body is never executed.
- The **do**-Statement tests the condition at the end of the loop. Therefore,
 - ◇ variables in the condition can still be initialized in the loop body,
 - ◇ the loop body is always executed at least once.

Do Statement (2)

- “do S while(C);” is equivalent to “ S while(C) S ”.

I.e. the statement S is first executed. Then the condition C is evaluated. If it is true, S is executed again, and C is evaluated again. And so on, until S is eventually false (i.e. 0).

- The do-statement is used much more seldom than the while-statement.

It is also a bit confusing that the same keyword `while` is used. A `while`-loop with an empty body is possible in C, since the loop condition can have side effects. In Pascal, the `do`-loop has the form “repeat S until C ”. However, there the loop ends when the condition becomes true (which is also confusing).

For Statement (1)

- The purpose of the `for`-statement is to execute a statement a specified number of times, where a variable (“loop variable”) steps through an interval.
- In Pascal, the `for`-statement has the form

```
for i := 1 to 20 do S      (* Pascal, not C *)
```

This will execute the statement S 20 times:

For $i = 1, i = 2, i = 3, \dots, i = 20$.

- The `for`-statement in C is more general: One can e.g. also step through the elements of a linked list.

For Statement (2)

- In C, the above `for`-statement is written as:

```
for(i = 0; i <= 20; i++) S
```

- In general “`for(A; B; C) S`” is equivalent to

```
A;          /* Initialization */
while(B) {  /* Loop Condition */
    S
    C;      /* Step */
}
```

- Advantage of the `for`-statement: The entire loop control is collected in one place, so it is easy to understand which values the variable will take.

For Statement (3)

- A `for`-statement to step through all elements of a linked list looks as follows:

```
for(p = anchor; p; p = p->next) S
```

- A `for`-statement to step through all characters of a string is written as follows:

```
for(p = str; *p; p++) S
```

- E.g. find the first non-space character in a string:

```
for(p = str; *p && isspace(*p); p++) ;
```

For Statement (4)

- All three parts of the **for**-statement are optional:

opt-expr → **expression**

opt-expr → ϵ

- If the first or last part is missing, the initialization or loop step must be done in some other way.

E.g. sometimes a parameter of a function is used as loop variable. Then no initialization is necessary (the function-call has assigned a value). E.g. `for(; *str; str++) S`.

- If the second part is missing, the loop is infinite, however, it can be stopped with **break** (see below).

For Statement (5)

- The loop variable must be declared before the loop.

In C++, it can be declared within the `for`.

- It can be accessed after the loop (then the value is the first value that made the loop condition false).
- C does not require that there is a uniquely determined “loop variable”.

E.g. sometimes one sees `for`-loops that manage two variables: One counter and one pointer that steps over an array. This can be done by using the comma-operator in the initialization and the loop step.

For Statement (6)

- It is legal but bad style to change the loop variable within the loop body.

Some languages (e.g. Pascal) explicitly forbid this. In some languages, the loop variable also has an undefined value after the loop (it is not clear whether it is the last value for which the loop body was actually executed or the first value that made the loop condition wrong).

- In languages like Pascal, an advantage of the `for`-loop over the `while`-loop is that termination is guaranteed for the `for`-loop.

In C this does not hold. However, the loop control should be sufficiently simple that termination is easy to verify.

Goto Statement (1)

- C has a `goto`-statement. One can put a label (an identifier) in front of any statement:

```
continue_here: x = x + 1;
```

- The label is automatically declared by using it.

There is no separate declaration of labels as in Pascal. Labels are valid for the current function. It is possible that the `goto` is written before the label is defined as long as both are in the same function.

- Then one can use the following statement to transfer control to the labelled statement:

```
goto continue_here;
```

Goto Statement (2)

- The `goto`-statement makes the semantics of programs more complicated to define and to understand:
 - ◇ It is no longer true that each statement has a single point of entry and a single point of exit.
 - ◇ Therefore, the state of a program must now also contain the current position in the program.

So if one starts with a single `goto` in an otherwise structured program, all constructs are formally explained in terms of `goto`.
 - ◇ The structure of a program is no longer a reliable guide for understanding it.

Goto Statement (3)

- Formally, the `goto`-statement is not necessary:
Even without `goto`, C is computationally complete.

“Computationally complete” means that any formal algorithm can be formulated in the language. Thus, any program written with `goto` can be translated into an equivalent one without `goto`. However, the equivalent program may be longer: It might be necessary to duplicate some code or to introduce new variables.

- Some style guides completely forbid the `goto`, some software companies allow their employees to use a `goto` only if they get a signature from a director.

“Although we are not dogmatic about the matter, it does seem that `goto`-statements should be used rarely, if at all.” (Kernighan/Ritchie).

Return Statement (1)

- A program is structured into functions.
Other names are: procedures, subprograms, subroutines, methods.
- It is possible to leave the current function with the “`return`”-statement.
- The `return`-statement is a form of `goto`, but a very structured one: The only place to jump to is the end of the current function.
Therefore, no label is required.
- Officially, Pascal has no `return` (some implementations added it). Java has `return`, but no `goto`.

Return Statement (2)

- The `return` construct is today accepted, although it also causes a transfer of control that violates the usual program structure.
- In C, there are two kinds of functions:
 - ◇ Functions that do not return a value (have the result type `void`): For them, one simply writes `return;` when the function is done.
 - ◇ Functions that return a value: Then one writes `return E;`, where `E` is the result value.

Return Statement (3)

- So the `return`-statement is also used to define the value of the function.

In Pascal, the function value is defined by an assignment to a variable that has the same name as the function. However, this variable may not appear in an expression (i.e. it is impossible to query its value).

- E.g. the function “`main`” should normally return 0:

```
return 0;
```

- At the end of the program code for each function, C implicitly inserts “`return;`”.

So when the end of the program code is reached, control is transferred to the caller. It is not necessary to write an explicit “`return;`” there.

Return Statement (4)

- If the function should return a value, the compiler may print a warning “control reaches end of non-void function” if there is no explicit return that defines a function value.

It might be necessary to increase the warning level, by default the compiler may not even issue a warning. The result value is undefined if the function falls off the end of the program text without `return E`. E.g., in one example I tried, I got “133224” as function value.

- The value of the expression after “`return`” is converted, as if by assignment, to the result type of the function.

Break Statement (1)

- Another special case of `goto` is one used to terminate a loop (or leave a `switch`).
- C has a special statement for this:

```
break;
```

The `break`-statement does not require a label.

- The `break`-statement transfers control to the first statement after the innermost loop (or `switch`) that encloses the `break`.

Break Statement (2)

Count the number of words (separated by spaces):

```
(1) char line[1000], *p;  
(2) int words = 0;  
(3) while(fgets(line, 1000, stdin) != 0) {  
(4)     p = line;  
(5)     while(*p) {  
(6)         while(*p && isspace(*p)) p++;  
(7)         if(!*p) break;  
(8)         words++;  
(9)         while(*p && !isspace(*p)) p++;  
(10)    }  
(11)    /* The break jumps here */  
(12) }
```

Break Statement (3)

- While the `goto`-statement is usually considered a significant violation of “good programming style”, the `break`-Statement is normal in C programs.

So the `break`-statement is a socially accepted kind of “structured goto”. It is slightly worse than the “`return`”, but not very much: If the loop were a procedure, it had the same effect.

- Of course, the `break`-statement also causes an unexpected transfer of control, and difficulties for a formal definition of program semantics.

Break Statement (4)

- The `break`-statement should not be used without need: One should invest some time to search for an elegant formulation without `break`.
- However, if the only alternative is to duplicate code, the `break` may be better.
- Especially, if the loop control is `while(1)` or `for(;;)`, it is obvious that there must be a `break` somewhere in the middle of the body.

A normal `for`-statement suggests that the entire loop control is shown at the top. Then it may be bad style to add a `break` in the body.

Continue Statement

- The statement “`continue;`” jumps to the end of the body of the current loop. If the loop condition is still true, the next iteration of the loop starts.

I.e. one jumps over the rest of the loop body only for the current iteration. One immediately continues with the next iteration. For the `for`-loop, the step/increment-expression is still evaluated (in this case, the `for` is not completely equivalent to the `while` shown above).

- “`continue`” is seldom used, maybe it was only introduced in analogy to “`break`”.

Since it is seldom used, it might be a bit confusing/unexpected. Therefore, it should be seldom used.

Summary: Jump Statements

- C has the following four kinds of jump statements:

jump-stmt → goto identifier ;

jump-stmt → break ;

jump-stmt → continue ;

jump-stmt → return opt-expr ;

- The standard library defines a procedure `exit(c)` to terminate the program. The return code *c* should be 0 for normal termination and positive for errors.

Switch Statement (1)

- The `switch`-statement is a conditional statement. It is used to consider several cases depending on the possible values of a variable (an expression), e.g.

```
(1)  switch(binop) {  
(2)      case op_plus:  
(3)          result = opd1 + opd2;  
(4)          break;  
(5)      case op_minus:  
(6)          result = opd1 - opd2;  
(7)          break;  
(8)      ...  
(9)  }
```

Switch Statement (2)

- The above program is similar to:

```
(1)  if(binop == op_plus)
(2)      result = opd1 + opd2;
(3)  else if(binop == op_minus)
(4)      result = opd1 - opd2;
(5)  else if ...
```

- However, the `switch` might sometimes be executed more efficiently, namely when there are many cases within a not too big interval of possible values.

E.g. when all possible values of an enumeration type variable are considered or many different values of a character variable.

Switch Statement (3)

- Then a good compiler will construct an array of start addresses of the program parts of the single cases.

At runtime, the processor will jump to the address in this array at the index position given by the `switch`-expression. So it is not necessary to compare the value of the `switch`-expression one by one with the possible cases.

- To make this implementation possible, C requires that the expression after `case` is a constant expression (so the compiler can compute its value).

Switch Statement (4)

- There is also a label “default” that can be used to specify an action for all other cases.

It corresponds to the final `else` in the `else if`-chain.

```
(1)  switch(binop) {
(2)      case op_plus:
(3)          result = opd1 + opd2;
(4)          break;
(5)      ...
(6)      default:
(7)          printf("Illegal operator!\n");
(8)          exit(1);
(9)  }
```

Switch Statement (5)

- It is not necessary to write “`default:`” at the end of the switch, although it is the usual place.
- It is no error if the `switch`-expression has a value different from all listed cases and there is no “`default`”.

Then simply nothing is done, just as if the final `else` in the `else if`-chain would be missing. However, a compiler may issue a warning if not all possible values of an enumeration type variable are considered and there is no “`default`” case. One can put a `default`-case with no code to make the compiler shut up.

Switch Statement (6)

- If a case does not end with “`break;`”, program execution continues with the next case.
- In this way, one can consider several values of the `switch`-expression in a single piece of program:

```
(1)  switch(next_char) {
(2)      case '0':
(3)      ...
(4)      case '9':
(5)          /* Integer */ ... break;
(6)      case 'a': ... case 'z':
(7)          /* Identifier */ ... break;
```

Switch Statement (7)

- However, this can also be abused for very tricky programming (which gives hard-to-read programs):

```
(1)  octal = 0;
(2)  switch(next_char) {
(3)      case '0':
(4)          octal = 1;
(5)      case '1':
(6)          ...
(7)      case '9':
(8)          /* Integer */ ... break;
(9)          ...
(10) }
```

Switch Statement (8)

- It is a common error to forget the `break` that terminates a `case`. Then the program code of the next case will also be executed.

Unfortunately, this is legal in C.

- Some compilers (and the semantic checker “`lint`”) may give warnings if a case has program code but no `break` at the end.

There might be special comments (e.g. “`/**FALLTHROUGH*/`”) or compiler directives (`#pragma`) to switch the warning off if one really wants this. But probably one should simply avoid such tricks.

Switch Statement (9)

- Actually, the “`case`” is like a label of a `goto`: It is even possible to place the `case` inside structured statements.

Of course, this is bad style. It is not much different from a `goto`.

- In summary, the syntax of labelled statements is:

labelled-stmt → `identifier` `:` statement

labelled-stmt → `case` `const-expr` `:` statement

labelled-stmt → `default` `:` statement