CTex Language Reference Manual

Weicheng Zhao, Rachel Liu, Unal Yigit Ozulku, and Hu Zheng

Lexical Conventions

Token

There are four types of tokens in CTex languages: identifiers, operators, constants and other symbols. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments as described below are ignored except as they serve to separate tokens.

Comments

The characters %% introduce a comment, which terminates characters %%. Comments do not nest.

Identifiers

An identifier could be:

- A single letter or a single Greek letter, for example, a and `\alpha`. Uppercase and lowercase letters and Greek letters are all supported. Upper and lower case (Greek) letters are considered different. No restriction on identifiers' length.
- Specific style operators with a sequence of letters or Greek letters between curly braces, for example, `\mathrm{abc}` and `\mathbb{\alpha\beta}`.
- Anything follows the two cases above with a underscore _ and a single digit, a single letter, a single Greek letter, a sequence of digits between curly braces or a sequence of letters or Greek letters between curly braces following. For example, `x_a` and `\mathbb{\alpha}_2`.

Empties are allowed between curly braces but unnecessary blanks will be removed, which means `\mathbb{\theta a}` and `\mathbb{\theta a}` are the same identifier.

Specific style operators that could be used in identifying an identifer in case 2 are:
`\mathrm` `\mathit` `\mathbf` `\mathsf` `\mathtt` `\mathfrak` `\mathcal` `\mathbb` `\mathscr`

Acceptable Greek letters in CTeX are as follows:
`\alpha` `\beta` `\Gamma` `\gamma` `\Delta` `\delta` `\epsilon` `\varepsilon` `\zeta` `\eta` `\Theta` `\theta` `\vartheta` `\iota` `\kappa` `\varkappa` Lambda `\lambda` `\mu` `\nu` `\Xi` `\xi` `\Pi` `\pi` `\varpi` `\rho` `\varrho` `\Sigma` `\sigma` `\varsigma` `\tau` `\Upsilon` `\upsilon` `\Phi` `\phi` `\varphi` `\chi` `\Psi` `\psi` `\Omega` `\omega`

All other expressions appearing after a backslash are not acceptable.

Constants

There are two kinds of constants, Integer Constant and Floating Constant.
An integer constant should consist a sequence of digits and would always be considered decimal. All integer constants will be considered as integer type.

A floating constant consits of an integer part, a decimal point and a fraction part. All floating constants will be considered as floating type.

Operators

Operators in CTeX are as follows.

^ _ ( ) / + - = < >
\cdot \times \div \sum \prod \frac \leq \geq \neq \mid \nmid \neg(for NOT) \binom \arccos \arcsin \arctan \cos \cosh \cot \coth \csc \exp \pmod \gcd \vee(for OR) \wedge(for AND) \lg \ln \log \sqrt \max \min \sec \sin \sinh \tan \tanh \|(for absolute value) \lfloor \rfloor \lceil \rceil

Other symbols

Besides of what is mentioned above, there are following symbols mostly used in CTeX language to seperate codes.

Symbol	Meaning
\\	Ends a statment or use as seperators between different cases
&	Seperator of case expression and then expression in the case statement
{}	Used to bracket parameters of some operators and a sequence of letters in definition of an identifier, also used in some expressions and form a statement closure
%	Print the result of the following expression
\begin{cases}	Starts the case statement
\end{cases}	Ends the case statement
,	Seperate argument list when defining or calling a function, also used in some experssions

Syntax

In this part, we will introduce expressions and their synatx in CTeX by giving out the formal definition of every kind of expressions.

Arithmetic conversions

When a description of an arithmetic operator below uses the phrase “the numeric arguments are converted to a common type”, this means that the operator implementation for built-in types works as follows:

If either argument is a floating point number, the other is converted to floating point;

otherwise, both must be integers and no conversion is necessary.

Some additional rules apply for certain operators.

Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom ::=  identifier | literal | parenth_form
```

Identifiers

An identifier occurring as an atom is a name. See section Identifiers for lexical definition. When the name is bound to an object, evaluation of the atom yields that object.

Literals

CTeX only supports integer and floating numeric literals:

```
literal ::=  integer | floatnumber
```

Evaluation of a literal yields an object of the given type with the given value. The value may be approximated in the case of floating point.

Parenthesized forms

A parenthesized form is an expression enclosed in parentheses or similiar math operators -- `\|` for absolute value, `\lfloor` `\rfloor` for floor and `\lceil` `\rceil` for ceiling:

```
parenth_form ::=
           "(" expr ")"
         | "\|" expr_calc "\|"
         | "\lfloor" expr_calc "\rfloor"
         | "\lceil" expr_calc "\rceil"
```

A parenthesized expression yields the single expression that makes up the expression list.

The power operator

The power operator binds more tightly than unary operators on its left, and binds less tightly than unary operators on its right. The syntax is:

```
expr_pow ::=  atom | atom "^" expr_unary | atom "^" "{" expr_calc "}"
```

The log-like function operators

The log-like function operators include `\lg` `\ln` `\log` `\sqrt` `\sin` `\cos` `\tan` `\arcsin` `\arccos` `\arctan` `\sinh` `\cosh` `\tanh` `\cot` `\sec` `\csc` `\coth`. They have the same priority. The synatx is:

```
log_like_ops ::= "\lg" | "\ln" | "\log"
| "\sqrt" | "\sin" | "\cos" | "\tan" | "\arcsin"
| "\arccos" | "\arctan" | "\sinh" | "\cosh"
| "\tanh" | "\cot" | "\sec" | "\csc" | "\coth"
log_op := "\log" UNDERLINE "{" expr_calc "}"
expr_log ::=  expr_pow | log_like_ops expr_pow | log_op  expr_pow
```

Unary arithmetic operations

All unary arithmetic operations have the same priority:

```
expr_unary ::=  expr_log | "-" expr_log | "+" expr_log
```

Implicit multiplication

Implicit multiplication declares the situation like x(x+y) in math. It's given higher priority than multiplicative operators.

```
expr_impl_mult ::= expr_unary | expr_impl_mult expr_unary
```

Multiplicative operators

The multiplicative operators * \codt \times / \div \pmod group left-to-right.

The * \codt \times operators yield the product of its arguments.

```
mult_op: "*" | "\cdot" | "\times"
```

The / \div operator yields the quotient of its arguments.

```
div_op: "/" | "\div"
```

The \pmod operator yields the the remainder from the division of the first expression by the second.

```
expr_mult:
    expr_impl_mult
    | expr_mult mult_op expr_impl_mult
    | expr_mult div_op expr_impl_mult
    | expr_mult "\pmod" expr_impl_mult
```

Additive operators

The additive operators + and − group left-to-right.

"expression + expression" yields the sum of the two expressions.
"expression - expression" yields the difference of the operands.

```
expr_add ::= expr_mult | expr_add "+" expr_mult | expr_add "-" expr_mult
```

Common operators

Common operators include \gcd \min \max. The synatx is:

```
com_op ::= "\gcd" | "\min" | "\max"
expr_com ::= com_op "(" expr_calc "," expr_calc ")"
```

Functional expressions

Functional expressions evaluate a function call that was defined before in the program by the user.

```
arg_list ::= expr_calc | expr_calc "," arg_list
expr_func ::= funcname "(" arg_list ")"
funcname ::= identifier
```

Frac-like operations

The frac-like function operators include `\frac` and `\binom`. The synatx is:

```
frac_op ::= "\frac" | "\binom"
expr_frac ::= frac_op "{" expr_calc "," expr_calc "}"
```

Large operators expression

The Large operators include `\sum` and `\prod`. They provide users an easier way to calculate some accumulation values. The synatx is:

```
large_op ::= "\sum" | "\prod"
expr_large_op ::= large_op "_" "{" index "=" start "}" "^" "{" end "}"
large_op_ expr
index ::= identifier
start ::= expr_calc
end ::= expr_calc
large_op_expr ::= expr_calc
```

The expression would evaluate following these steps:

1. Evaluate the expression "start".
2. Assign the result to the identifier "index", if "index" is not defined before in the same scope then "index" would become a local variable whose scope is only in this expression, otherwise, "index" would be rebound.
3. Evaluate the expression "end".
4. Check if "index" is greater than the result, if so, go to step 8.
5. Evaluate the expression "large_op_expr" and store the results
6. Increase the "index" by 1.
7. Go to step 3.
8. Accumulate all the stored results accounding to the type of the operations and take the result as the result of this expression.

Calculating expressions

Calculating expressions are illogical expressions, which calculate the value of the expressions.

```
expr_calc ::= expr_add | expr_com | expr_func | expr_frac | expr_large_op
```

Comparisons

All comparison operations in CTex have the same priority, which is lower than that of any arithmetic operations.

```
expr_comp ::=
    expr_calc
    | expr_comp "<" expr_calc
    | expr_comp ">" expr_calc
    | expr_comp "\leq" expr_calc
    | expr_comp "\leq" expr_calc
    | expr_comp "=" expr_calc
    | expr_comp "\neq" expr_calc
    | expr_comp "\nmid" expr_calc
    | expr_comp "\mid" expr_calc
```

Logical Expressions

In CTex, boolean operations are evaluated from left to right, but they do not yield outputs.

```
expr_logic  ::=
      expr_comp
    | expr_logic "\wedge" expr_comp
    | expr_logic "\vee" expr_comp
    | "\neg" expr_comp
```

All Expressions

All expressions in CTeX can be defined as

```
expr ::= expr_logic
```

Operator Precedence

The precedence of expression operators is the same as the order of the following table (highest precedence first):

```
atom:
    IDENTIFIER
    | INT_CONST
    | FLOAT_CONST
    | "(" expr ")"
    | "\|" expr_calc "\|"
    | "\lfloor" expr_calc "\rfloor"
    | "\lceil" expr_calc "\rceil"

expr_pow:
    atom
    | atom "^" expr_unary
    | atom "^" "{" expr_calc "}"

log_like_op:
     "\lg" | "\ln" | "\log" | "\sqrt" | "\sin" | "\cos" | "\tan"
     | "\arcsin" | "\arccos" | "\arctan" | "\sinh" | "\cosh"
     | "\tanh" | "\cot" | "\sec" | "\csc" | "\coth"
log_op:
    "\log" "_" "{" expr_calc "}"
expr_log:
    expr_pow | log_like_ops expr_pow | log_op  expr_pow

expr_unary:
    expr_pow
    | expr_log
    | "+" expr_pow
    | "-" expr_pow

expr_impl_mult:
    expr_unary
```

```
    | expr_impl_mult expr_unary

mult_op: "*" | "\cdot" | "\times"
div_op: "/" | "\div"
expr_mult:
    expr_impl_mult
    | expr_mult mult_op expr_impl_mult
    | expr_mult div_op expr_impl_mult
    | expr_mult "\pmod" expr_impl_mult

expr_add:
    expr_mult
    | expr_add "+" expr_mult
    | expr_add "-" expr_mult

com_op: "\gcd" | "\min" | "\max"
expr_com: com_op "(" expr_calc "," expr_calc ")"

arg_list: expr_calc | expr_calc "," arg_list
expr_func: identifier "(" arg_list ")"

frac_op: "\frac" | "\binom"
expr_frac: frac_op "{" expr_calc "," expr_calc "}"

large_op: "\sum" | "\prod"
expr_large_op: large_op "_" "{" identifier "=" expr_calc "}" "^" "{" expr_calc
"}" expr_calc

expr_calc:
    expr_add | expr_com | expr_func | expr_frac | expr_large_op

expr_comp:
    expr_calc
    | expr_comp "<" expr_calc
    | expr_comp ">" expr_calc
    | expr_comp "\leq" expr_calc
    | expr_comp "\leq" expr_calc
    | expr_comp "=" expr_calc
    | expr_comp "\neq" expr_calc
    | expr_comp "\nmid" expr_calc
    | expr_comp "\mid" expr_calc

expr_logic:
    expr_comp
    | expr_logic "\wedge" expr_comp
    | expr_logic "\vee" expr_comp
    | "\neg" expr_comp

expr:
    expr_logic
```

Statements

There are 6 kinds of statements in CTeX language: expression statement, assignment statement, print statement, function definition statement and case statement. We call first 3 kinds of statment as simple statment, while the last 3 kinds as complicated statment. A statement ends with double backslashes. Statements can compound together into a statement list within curly braces as a single statment to be used in complicated statement.

```
simple_stmt ::= expression_stmt | assignment_stmt | print_stmt
comp_stmt ::= func_def | case_stmt
single_stmt ::= simple_stmt | comp_stmt
stmt_list ::= single_stmt | single_stmt "\\" stmt_list
stmt_closure ::= "{" single_stmt "\\" stmt_list "}"
stmt ::= single_stmt | stmt_closure
stmts ::= stmt | stmt "\\" stmts
```

Scope

If and only if the assignment statment or function definition statement is a part of a complicated statement and the identifier has not been bound outside the complicated statement, the assignment statement or function definition statement will make the identifier local. Otherwise, it will make the identifier be able to use and refer in the whole program globally, even in other complicated statments.

Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt ::= expr_calc
```

An expression statement evaluates the expression.

Assignment statements

Assignment statements are used to (re)bind names to values:

```
assignment_stmt ::=  identifier "=" expr_calc
```

An assignment statement evaluates the expression and assigns result to the identifier. The identifier would be rebound if it was already bound.

Print statements

Print statements are used to print the evalution result of an expression to standard output. It will also output a newline \n implicitly after outputing the result.

```
print_stmt ::= "%" expr_calc
```

Function definitions

A function definition defines a user-defined function.

```
func_def ::= funcname "(" param_list  ")" "=" stmt
param_list ::= identifier | identifier "," param_list
funcname ::= identifier
```

A function definition is an executable statement. Its execution binds the function name. The function definition does not execute the function body.

Case statements conditional execution

The case statement is used for conditional execution

```
case_stmt ::= "\begin{cases}" case_stmt_list "\end{cases}"
case_stmt_list ::= suite case_stmt_list
suite ::= stmt "&" expr_logic "\\"
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true; then the statement in that suite is executed and no other part of the case statement is executed or evaluated. If all expressions are false, then none of the suites would be executed.

Top Level and Full Grammar specification

The CTeX compiler will get its input from the file. The full grammar of CTeX are as follows.

```
file: [stmts] EOF

stmts:
    stmt "\\" stmts*
stmt:
    single_stmt
    | stmt_closure
stmt_closure:
    "{" single_stmt "\\" stmt_list "}"
stmt_list:
    single_stmt ("\\" stmt_list)*
single_stmt:
    expression_stmt
    | assignment_stmt
    | print_stmt
    | func_def
    | case_stmt

expression_stmt: expr_calc
assignment_stmt: identifier "=" expr_calc
print_stmt: "%" expr_calc
func_def: identifier "(" param_list  ")" "=" stmt
param_list: identifier ("," param_list)*
case_stmt: "\begin{cases}" case_stmt_list "\end{cases}"
case_stmt_list: suite case_stmt_list
suite: stmt "&" expr_logic "\\"
```