Introduction

This is the Language Reference Manual for CHRONOS (CHRONologically Organized Scheme). Chronos was developed with the intent of providing college students with a tool to organized the many courses they will take into a schedule that fits their many needs. The manual presented here is supplied with the hopes that it will allow the user to utilize Chronos as effectively as possible.

Lexical Convention

A program in Chronos consists of a single *program* (See Grammar) in a single CHRONOS file. The translation of the source language to the target language contains many stages. The first involves scanning in the tokens from the sources code.

Tokens

There are six classes of token: identifiers, keywords, constants, string literals, operators, and other separators. All types of whitespace (spaces, horizontal, newlines, formfeeds) are ignored, as are comment characters and the tokens that occur between them.

Comments

There are two types of characters that allow for comment:

|  |  |
| --- | --- |
| Comment Character | Description |
| // | This character begins a comment terminated by a newline  Ex: // this is a comment |
| /\* | This character begins a comment that can span multiple lines, and is terminated by \*/  Ex: /\*  this comment that spans  more than one line  \*/ |

NOTE: Including these comment characters inside a string doesn’t produce a comment, i.e.

“This is a string /\* not a comment \*/” will be read as a string.

Identifiers

An identifier is a sequence of letters and digits that can also included underscores. The identifier can begin with either a letter or an underscore, but not a digit. While both upper and lower case letters are permitted, Chronos is case-sensitive, and will therefore distinguish between ‘this\_var’ and ‘this\_VAR’. As there is no limit on the length of an identifier, ‘this\_is \_the\_longest\_and\_most\_obnoxious\_identifier\_written\_in\_any\_prog\_language’ is a very possible identifier although, for practical reasons, we would not recommend it.

Keywords

The following is a list of reserved word and built-in function names for our language which cannot be used for any other purposes:

|  |  |
| --- | --- |
| Reserved Words | Built-In Function Names |
| |  |  | | --- | --- | | if | CourseList | | else | Timeblock | | new | Datetime | | foreach | int | | in | double | | break | Dayblock | | Schedule | Time | | Course | string | | |  |  | | --- | --- | | getDateTime | numCourses | | getDayblock | getCourse | | getTimeblock | conflicts | | getEndTime | has | | getStartTime | toString | | numSections | equals | | add | compareTo | | numCredits | lessThan | | greaterThan |  | |

Constants

There are two kinds of constants, which each associated with a specific data type. They are listed in the following grammar fragment:

constant

: INT

| FLOAT

;

Integer Constant

An integer constant is defined to be a sequence containing only digits 0 through 9, without a decimal point or fractional parts. Given the applications of Chronos, hexadecimal or octal representations of integers are not recognized.

The following are valid and invalid representation of integer constants:

|  |  |
| --- | --- |
| Valid Integer Representation | Invalid Integer Representation |
| |  | | --- | | 014 | | 25 | | 100 | | |  | | --- | | 100. | | 25.00 | | 32.5 | |

Integer constants are associated with data type int.

Floating Constants

Just like integer constants, floating constants are defined to be a sequence of digits 0 through 9, but unlike integer constants they can contain a decimal point and a fractional part. In addition to this, an exponential component may be added to the aforementioned representation of the floating constant (known as the significant digits) for scaling.

The following are all valid and invalid representations of floating constants:

|  |  |
| --- | --- |
| Valid Floating Representation | Invalid Floating Representation |
| |  | | --- | | 100. | | 2.718 | | 0.5 | | 1.23e5, 1.23e-5, 1.23e05, 1.23e-05 | | 1.23E5, 1.23E-5, 1.23E05, 1.23E-05 | | 7e3 | | |  | | --- | | 2.1.2 | | 1..9 | | e7 | |  | |

Floating constants are associated with data type double.

String Literals

A string literal, also known as a string constant, is a sequence of characters enclosed by double quotation marks, such as “this is a string”, and a variable declared as type string is initialized to these characters. Certain characters such as double quotation marks and newlines cannot appear within string literals in their expected, normal form, and therefore need escape sequences. The following are the escape sequences recognized by Chronos:

|  |  |
| --- | --- |
| Character Name | Escape Sequence |
| |  | | --- | | Backslash | | Backspace | | Tab | | Newline | | Formfeed | | Carriage-return | | Double Quotation | | Single Quoatation | | |  | | --- | | \\ | | \b | | \t | | \n | | \f | | \r | | \” | | \’ | |

Syntax Notation

Code snippets are presented in Courier New font.

Meaning of Identifiers

In Chronos, identifiers refer to the names of functions and objects of type int, double, course, courselist and schedule. These objects refer to locations in memory and have two main attributes: storage class and type. The storage class refers to the lifetime of the object while the types determine how the object is interpreted by the language.  
  
Because the object is static, it exists for the entire duration of the code being executed and its value is retained upon entry into and exit from blocks and functions.  
  
Basic Types  
  
The basic types are int, double, string.  
  
Those familiar with programming will immediately recognize these basic types. The int type. As with other programming languages such as C and Java, the size, and therefore the range, of this data is machine architecture dependent. That is to say, the values the int type can take on a 32-bit machine will differ from the values it can take on a machine with 64-bit architecture.  
  
As expected, the double type is encodes a floating point number in 64-bits.  
  
The string data type is a sequence of characters that are enclosed in double quotation marks.  
  
  
Derived Types  
  
The derived types are time, timeblock, dayblock, datetime, course, courselist, and schedule.

The time data type is used to represents the time of day and its representation is specifically formatted to the 24-hour clock. That is to say, when using the time type, the programmer should type is as (h)h:mm. The range of this type is 00:00 (or 0:00) to 23:59.

timeblock represents a span of time, with a start time and an end time.

dayblock represents the subset of the days of the week (Monday through Friday).

datetime contains both a timeblock and a dayblock.

An object of type course embodies all the information essential to describing a real-world course and relevant to this class scheduling program language. It aggregates the times and days that a course meets and the number of points the course is worth.

An object of type courselist is a list of courses.

Lastly, a schedule type is also a list of courses, with the only difference being that a courselist is allowed to have conflicts (time-wise), while an object of type Schedule is not.

The table below summarizes the derived types and their members.

|  |  |
| --- | --- |
| Derived Types | Their members |
| time | int time |
| timeblock | Time start  Time end |
| dayblock | char[] days |
| datetime | Dayblock db  Timeblock tb |
| course | string name  double credits  int selectedSection |
| courseList | string name |
| schedule | string name |

Objects and Lvalues  
  
As is the case with many programming languages, in Chronos, for most of the data tyupes, there is strong tie between objects and lvalues. Speaking more specifically, the object is some named region in memory and the lvalue is some expression referring to this object. An example of code using the conventional concept of the lvalue is as follows:  
          
 *exp1 = exp2;*  
  
Where *exp1* is referred to as the lvalue because it appears to the left of the assignment operator, and *exp2* is referred to as the rvalue because it appears to the right of the assignment operator.

Conversions

At the time of this manual’s publication, Chronos does not support type conversion.

Expressions  
  
In Chronos, an expression is any valid combination of variables, operators, constants, functions and explicit values that are interpreted by the Chronos compiler and generates another value.  
  
Each of the types above is described in more detail below.  
  
  
Primary Expressions  
  
Primary expression is an identifier, constant, string, or an expression in parentheses and they can take one of the three forms:

|  |  |  |
| --- | --- | --- |
| Identifier | Constant | String |

An identifier is a combination of alphanumeric characters, the first being the letter of an alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.  In this vein, the following are valid and invalid identifiers:

|  |  |
| --- | --- |
| Valid Identifiers | Invalid Identifiers |
| |  | | --- | | Math | | Math\_1\_1001 | | 1\_Math (because the first character is not a letter or underscore) |

An identifier is a primary expression provided it designates an object or a function.  
  
For example, fall2012.add(mathv1101), which adds a course called mathv1101 to the semester object contains two identifiers that are also primary expressions. We have fall2012 which is an identifier that designates and object of type Schedule and add which is an identifier that designates a function, also known as the *function designator*.

A constant is a primary expression and its type is determined by its form and value.  
  
A string literal is a primary expression. In the code following line of code,

electives.add(CL.getCourse("PE"));,

which adds a Physical Education course to the courselist object names electives, the string literal “PE” is a primary expression.  
  
  
Postfix Expression and Operators  
  
A postfix expression consist of the above mentioned primary expressions and expressions followed by postfix operators. These operators consist of:

Function Call Operator

A postfix expression followed by a function call operator denotes a function call. This is illustrated in the following grammar fragment:

postfix\_expr

: primary\_expr function\_parens?

;

function\_parens

: '(' argument\_expr\_list? ')'

;

argument\_expr\_list

: (expr) (',' expr)\*

;

The arguments to the function call operator can be zero or more expressions; if there is more than one argument, they are separated by commas. In the follow code fragment,

fall2012.add(mathv1101),

add is a primary expression, closely followed by the parentheses which contains a list of on argument.

Member Access Operator:

The member access operator ‘.’ is used to make reference to members of classes. For example,

fall2012.add(mathv1101).

Here, fall2012 is an object of type schedule. There exists an in-built function in the schedule data type that allow users to add classes to any schedule object. The member access operator enables fall2012 to access this member function add.  
   
The operators in postfix expressions are left-associative.

Other Expressions and Operators

Assignment Expression and Operator  
  
There is only one assignment operator and that is the ‘=’ operator and it groups from right-to-left. This assignment operator must have a modifiable value on its left side. This value by definition cannot be a constant value. In the simple assignment with ‘ =’, the value of the expression replaces the value of the object referred to by the lvalue. For example:

|  |  |
| --- | --- |
| Valid Assignment Expression | Invalid Assignment Expression |
| int courseMax = 6 | 3 = int courseMin |

In valid assignment expression above, the courseMax is an lvalue that is assigned a new rvalue of 6.

Comma Operator  
  
The comma operator is primarily used for explicitly partitioning the elements of a list or array. Specifically, the comma operators are used to separate an argument-list of a function call that has more than one element. It is also used to separate the character ‘M’, ‘T’, ‘W’, ‘R’, and ‘F’ that correspond to the five business days of the week (Monday through Friday, respectively).

We have already seen an argument-list for a function, but for array of characters, the following grammar fragment illustrates the proper notation:

dayblock

: '[' DAY ( ',' DAY )\* ']'

;

Arithmetic Operators

There are two groups of arithmetic operators known as additive operators and multiplicative operators, both of which are left-associative. In Chronos, the two members of the additive operator group are the addition operator (+) and the subtraction operator (-), and its grammar fragment is as follows:

math\_expr

: math\_term ( ('+' | '-') math\_term )\*

;

The two members of the multiplicative operator group are the multiplication operator (\*) and the division operator (/). The grammar fragment for this arithmetic operator group is shown below:

math\_term

: unary\_expr ( ('\*' | '/') unary\_expr )\*

| timeblock

;

All arithmetic operators behave as one would expect, i.e. the addition operator, subtraction operator, multiplication operator, and division operator perform addition, subtraction, multiplication and division, respectively.

Logical Operator

Logical operators compare logical expressions and in turn produce a value that can be interpreted as true or false. There are two logical operators: the logical-AND operator (&&) and the logical-OR operator (||). The logical-AND operator produces a true value if both its operands evaluate to true, otherwise it produces a false value. The logical-OR operator produces a true value if at least one of its two operands are true, otherwise it evaluates to false.

Relational Operators

Relational operators compare the values of variables in a relational expression and, like logical operators, produce a value that can be interpreted as either true or false. The relation operators in Chronos are the less-than operator (<), the greater-than operator (>), the less-than-or-equal-to operator (LEQ), and the greater-than-or-equal-to (GEQ). These operators acts one would expect, i.e. int x < int y is true if x is less than y, false otherwise. The other relational operators behave similarly. What follows is a grammar fragment for the relational operators:

rel\_expr

: math\_expr ( ('<' | '>' | GEQ | LEQ) math\_expr )\*

| datetime

;

Equality Operators

The equality operators work similarly to the relational operators, but it instead specifically checks whether or not the values of two variables are equal. The two equality operators are the equal operator (EQ) and the not-equal operator (NEQ), and the grammar for the equality operator is:

equiv\_expr

: rel\_expr ( (EQ | NEQ) rel\_expr )\*

;

Many of the previous operators are overloaded, and therefore can applied to derived data types. The following table summarizes what operators can applied to each type.

|  |  |
| --- | --- |
| Data Type | Operators |
| int | +, -, \*, /, <,>, LEQ, GEQ, EQ, NQ |
| double | +, -, \*, /, <,>, LEQ, GEQ, EQ, NQ |
| string | EQ, NEQ, + |
| time | <,>, LEQ, GEQ, EQ, NQ |
| timeblock | EQ, NQ |
| dayblock | EQ, NQ |
| datetime | EQ, NQ |
| course | EQ, NQ |
| courselist | EQ, NQ |
| schedule | EQ, NQ |

With the exception of the (+) operator for the string data type, the operands of binary operators must be of the same type.

Declarations  
  
Declarations determine how each object is interpreted by the language and have the following form:

declarator:

type\_specifier ID

The type\_specifier is the type being used, and ID is an identifier. The grammar fragment for a type\_specifier is shown below:  
  
 type\_specifier

: INT\_T

| DOUBLE\_T

| DAYS\_T

| TIME\_T

| STRING\_T

| SCHEDULE\_T

| COURSE\_T

| COURSELIST\_T

| TIMEBLOCK\_T

| DATETIME\_T

;

Storage Class Specifiers  
  
Since there is only one default storage class (static), there is no storage class specifier.  
  
  
Type Specifiers  
  
The type specifiers are:  
          
 int  
 double  
 string  
 time

timeblock

dayblock

datetime  
     course  
     courselist  
     schedule  
  
  
Initialization  
  
Basic data types are initialized with the assignment operator.  
  
Derived data types are first declared, then initialized via the built-in functions that correspond to the particular derived type.

Statements

With the exception of jumps due to conditionals, statement are executed in sequence and fall under the categories listed in the grammar fragment below:

stmt

: expr';'

| selection\_stmt

| iteration\_stmt

| jump\_stmt';'

| ';'

;

Expression Statements

An expression statement is consists of an expression (See section entitled *Expression*) terminated with a semicolon.

Selection Statements

Selection statements determine the control flow of a program’s execution. There is only one selection statement, the if statement, and it takes on two forms:

if(*expression*)

*stmt*

and

if(*expression*)

*statement1*

else

*statement2*

In the first form, the expression is evaluated and if true, the following *stmt* is executed. If the expression evaluates to false, then the following *stmt* is not executed. The behavior of the second form is similar, except if expression evaluates to true, then *statement1* is executed; otherwise *statement2* is executed.

Iteration Statements

Iterations statements specify looping. The foreach statement is the only iteration statement. It takes the form

iteration\_stmt

: FOREACH\_T COURSE\_T element=ID IN\_T list=ID '{' line\* '}'

;

The COURSE\_T is restricted to type Courses, though the list can be either of type courselist or schedule. The iteration statement will cycle through all the elements in ‘list’, and at each element executed ‘line’.

Jump Statements

A jump statement unconditionally transfers the control flow of execution. The only jump statement in Chronos is the break statement, as indicated by the grammar fragment of the jump statement:

*jump\_stmt*

*break;*

The break statement can only appear in an iteration statement, i.e. a foreach statement.

Built-In Functions

There are two built-in function not associated with any type. They are:

|  |  |
| --- | --- |
| void print(string str) | void creatCourseList(string str) |

The print method takes a string as an argument and outputs it to console. All types can be concatenated to type string. createCourseList takes as an input the name of a text file containing the name, and other information of courses. The file must be specially formatted, and depending on the current working directory, a path, in addition to the file name, may have to be provided as the string input to the method.

The follow is a list of the built-in functions associated with each derived data type.

**time:**  
public int lessThan(time t)  
public int greaterThan(time t)    
  
**timeblock:**  
public int conflicts(timeblock tb)  
  
**dayblock:**  
public int has(char db)  
public int add(char ch)    
  
**datetime:**

public string toString()

public int conflicts(datetime dt)

**course:**  
public datetime getDatetime()  
public dayblock getDayblock()  
public timeblock getTimeblock()  
public time getStartTime()  
public time getEndTime()  
public int numSections()  
  
**courselist:**  
public void add(course c)  
public int numCredits()  
public int numCourses()  
public int numCourses(dayblock db)  
public course getCourse(string str)

**schedule:**  
public int numCredits()  
public int numCourses()  
public int numCourses(dayblock db)  
public Course getCourse(string str)   
public int add(course c)

Grammar

// starting rule

program

: line+ EOF

;

line

: declarator

| stmt

;

declarator

// matches int x;

: type\_specifier ID ';'

// matches int x = 5;

| type\_specifier ID '=' expr ';'

;

stmt

: expr';'

| selection\_stmt

| iteration\_stmt

| jump\_stmt';'

| ';'

;

selection\_stmt

// matches if and if/else statements

: IF\_T expr '{'(a=line)\* '}' (ELSE\_T '{' (b=line)\* '}')?

;

iteration\_stmt

// matches foreach statements

: FOREACH\_T COURSE\_T element=ID IN\_T list=ID '{' line\* '}'

; // iterations only exist for courses

jump\_stmt

// matches break

: BREAK\_T

;

expr

// matches OR statements or assignment expressions

: and\_expr (OR and\_expr)\*

| assignment\_expr

;

assignment\_expr

// matches x = y, y = 5, z = 5 \* 3, etc

: ID '=' expr

;

and\_expr

// matches AND statements

: equiv\_expr (AND equiv\_expr)\*

;

equiv\_expr

// matches equivalence relations

: rel\_expr ( (EQ | NEQ) rel\_expr )\*

;

rel\_expr

// matches other relations

: math\_expr ( ('<' | '>' | GEQ | LEQ) math\_expr )\*

| datetime

;

math\_expr

// matches add/subtract expr

: math\_term ( ('+' | '-') math\_term )\*

;

math\_term

// matches mult/division expr

: unary\_expr ( ('\*' | '/') unary\_expr )\*

| timeblock

;

unary\_expr

: postfix\_expr ('.' postfix\_expr)\*

| NOT postfix\_expr

;

postfix\_expr

// matches functions or variables

: primary\_expr function\_parens?

;

function\_parens

// matches () and the params in a function call

: '(' argument\_expr\_list? ')'

;

datetime

// matches [M,W] 10:00~11:00

: dayblock timeblock

;

timeblock

// matches 13:00~14:00

: a=TIME '~' b=TIME

;

dayblock

// matches [M,W,F] etc

: '[' DAY ( ',' DAY )\* ']'

;

primary\_expr

: constant

| ID

| STRING

| TIME

| '('expr')'

;

argument\_expr\_list

: (expr) (',' expr)\*

;

constant

: INT

| FLOAT

;

type\_specifier

: INT\_T

| DOUBLE\_T

| DAYS\_T

| TIME\_T

| STRING\_T

| SCHEDULE\_T

| COURSE\_T

| COURSELIST\_T

| TIMEBLOCK\_T

| DATETIME\_T

;