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 | days | | schedule | time | | course | string | | |  |  | | --- | --- | | name | getCourse | | numCredits | add | | conflict | print | | getDays |  | | getTimeBlock |  | | getStartTime |  | | getEndTime |  | | numCourses |  | |

Constants

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

*constant:*

*integer-constant*

*floating-constant*

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 | |

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 | |  | |

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 (may not be necessary)

Meaning of Identifiers

In Chronos, identifiers refer to the names of functions and objects. to integers, doubles, course, courseList and schedule objects. 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 day, time, int, double, string.  
  
The day type denotes the day of the week, excluding Saturday and Sunday. Since the focus of the language is to schedule classes, we believe this to be an appropriate omission as it is rare for an academic institution to have classes held on Saturdays and Sundays. The day type is essentially a single character that can take one of five values. These values are:  
  
M : this value of day represents Monday  
T   : this value of day represents Tuesday  
W : this value of day represents Wednesday  
R   : this value of day represents Thursday  
F   : this value of day represents Friday  
  
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 hh:mm. The range of this type is 00:00 to 23:59. Unlike the day type, no assumptions were made as to what times would and would not be appropriate to hold classes.  
  
More familiar to the average programmer is the int type. As with other programming languages such as C and Java, this 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.

These types were named so that the user can have an intuitive understanding of the types through just the name alone. With that being said, the Time type is the representation of time with the 24-hour format. Timeblock is 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 Timeblocka 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 data of the times and days (via ArrayList) that a course meets and the number of points the course is worth.

An object of type *CourseList* is a list of *course*s.

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 a object of type Schedule is not.

The table below summarizes the derived types and their members.

|  |  |
| --- | --- |
| Derived Types | Their members |
| Time | * int hr * int min |
| Timeblock | * Time start * Time end |
| Dayblock | ArrayList<String> days |
| DateTime | * DayBlock d * Timeblock t |
| Course | * string name * double points * ArrayList<Datetime> sections * int selectedSection |
| CourseList | * string name * ArrayList<Course> courses |
| Schedule | * string name * ArrayList<Course> courses |

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. **(May have to consider arithmetic 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.  
  
Every expression in name consists of one or more operators and at least one operand. Statements like:

* max points = 18
* max courses(all) = 5

are considered expressions.  
  
Technically speaking, NAME has the following types of expressions:

* Primary Expressions
* Postfix Expressions
* Assignment Expressions
* Comma Operator

Each of the types above is described in more detail below:  
  
  
Primary Expressions  
  
Primary expressions are identifiers, constants, strings, or expressions in parentheses and they can take the following 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 identifiers:

* Math
* Math\_1\_1001

While the following is not a valid identifier

* 1\_French (Because the first letter is not a letter of the alphabet).

An identifier is a primary expression provided it designates an object or a function.  
  
For example,

* The term ‘fall2012.add(math-v1101, 8)’ which adds a course called ‘math-v1101’ to the semester object contains two identifiers that are also primary expressions. We have:
  + ‘fall2012’ which is an identifier that designates the object for the term ‘Fall 2012’
  + ‘add’ which is an identifier that designates a function, also known as the *function designator*.

A constant is a primary expression. Its type is determined by its form and value.  
  
A string literal is a primary expression. In the code to create a new schedule for the fall semester of 2012, ‘create schedule(fall2012, “Fall 2012”)’ from our sample code, the string literal “Fall 2012” is a primary expression.  
  
  
Postfix Expression and Operators  
  
In NAME, postfix expression consist of the above mentioned primary expressions and expressions followed by postfix operators. In NAME, 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-expression ( [argument-expression-list ] )*.

The arguments to the function call operator are two or more expressions separated by commas. Examples of postfix expressions with function call operators from NAME are below:

fall2012.add(math-v1101, 8)

In these examples (which are different stages in creation of the object for the Fall 2012 semester),   the terms ‘add’ and ‘make’ are the primary expressions and they are closely followed by the parentheses which contains the comma separated expression list in the first, and the lone expression in the second.

Member Access Operator:

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

fall2012.add(math-v1101, 8)

Here, the 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 the fall object to access this member function ‘add’.

Derived Data Input Operator

In NAME, there are a number of derived data types with parameter inputs. They are constructed in the same way as Function Call Operators and are structures like so,

postfix-expression ( [argument-expression-list ] )

These include the derived data types:

* + Courses
  + CourseList
  + Schedule

Example code to illustrate this concept is as follows:

The operators in postfix expressions group left to right.

Other Operators

Assignment Operator  
  
There is only one assignment operator in the NAME language 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 that of the object referred to by the lvalue. For example:

* max points = 18; // maximum number of points per schedule
* max courses(all) = 5; // max number of courses per schedule

In both examples above, the values on the right hand side (‘18’ and ‘5’ respectively) are being assigned to the lvalues on the left hand side.

Comma Operator  
  
The comma operator in NAME exists in the two derived data types:

* courses
* CourseList

It works by storing the information from every argument beyond the first in the first arguments. For example, in the creation of groups, we have:

Here the comma operator works by adding the values ‘math-v1101’ and ‘stat-2102’ to the  primary expression ‘cs\_core’, such that whenever cs\_core is called throughout the program, we know that it is referring to the two classes ‘math-v1101’ and ‘stat-2102’.

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:

**[Grammar Fragment]**

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:

**[Grammar Fragment]**

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.

Relational Operators

Relational operators compare the values of variables in a relational expression and in turn 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). What follows is a grammar fragment for the relational operators:

**[Grammar Fragment]**

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:

**[Grammar Fragment]**

The operators mentioned in the *Other Operators* sectioncan only be performed on integers or floating point numbers, i.e. objects of type int or double, and the operands must be of the same type. It should also be noted that these operators cannot be overloaded and applied to the other data types mentioned previously.

Below is a table summarizing the operators and their properties.

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

*declarator:*

*type\_specifier identifier*

The type-list contains the types being used, and type refers to the basic types described earlier.  
  
        *type-list:*

*type*

*type, type-list*

*(type-list)*

*[type-list]*

*type:*

*integer-constant*

*day-constant*

*time-constant*

*floating-constant*

*string-literal*

Storage Class Specifiers  
  
Since there is only one default storage class (static), there is no storage class specifier.  
  
  
Type Specifiers  
  
The type specifiers are:  
          
        day  
 time  
 int  
 double  
 string  
 *point*  
*course*  
*group*  
*schedule*  
  
  
Initialization  
  
In NAME, derived data types are initialized when they are declared. Therefore, the form for initialization follows that of from section A.8.  
  
For the basic data, which are not declared, only initialized, initialization occurs by typing the value of the data type explicitly, following the appropriate format of the type (such as hh:mm for type time).

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:

*Statement*

*exp\_stmt*

*selection\_stmt*

*intertation\_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

foreach *data\_type identifier* in *list*

*stmt*

The *data\_type­* 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 *stmt*.

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