Amit Shravan Patel
UNI: ap3567
COMS W4115 (CVN)
Prof. Stephen Edwards
Language Reference Manual

AP++

1. Introduction

AP++ is designed to be a lightweight language for writing basic programs, with a particular emphasis on a list syntax and functionality. It consists of a subset of stylistic and syntactic elements from C, along with Python-inspired list support.

2. Lexical Conventions

2.1 Whitespace

Spaces, tabs, return and newlines are ignored during token parsing.

2.2 Comments

Comments are enclosed in /* */ and can span multiple lines. Nesting comments is not supported, e.g. /* /* this is /* not */ valid */ */.

2.3 Reserved words

The following are reserved words and cannot be used as identifiers:

if elseif else while break continue return int bool

2.4 Code Structure

Indentation is not required to parse *AP++*, but it is highly recommended for producing clean, manageable code. Instead, much like C, statement blocks are enclosed by {} braces and statements are terminated by semicolons.

3. Variables

Variables are names that are used to refer to some location in memory.

3.1 Primitives

Variables have two primitive types: {*bool, int*}.

bool variables can have two possible values {true, false}
int variable have max/min 32-bit integer values [-2147483647, 2147483647]

3.2 Variables are referred by identifier names of unbounded length, which take the regex form: ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0-9' '_], i.e. must start with alphabetical characters following by alpha-numeric characters or underscore.

e.g.
Valid: var, var1, var_, Var, v4r_

Invalid: 1var, .var, _var

3.3 Declaration

Variables are declared by the type followed by identifier name and semicolon:

```
int x;
bool y;
```

3.4 Literals

Literals are fixed value constants that do not alter during execution. Variables can be assigned literal values.

bool literals have the format {true, false}
e.g. bool b1 = true; bool b2 = false;

int literals have the format: [0-9]+ from [MIN_INT, MAX_INT]
e.g. int num1 = 1; int num2 = -1;

3.5 Assignment

In addition to literal assignment, variables can be assigned to other variables as long as that variable is in scope (see Section 3.6). Assignment can occur at declaration time or as a standalone statement. Assignment of primitive typed variables occur by value and are copied to the destination variable's address.

e.g.

```
int x = 3;
int y = x;
x = 4;
/* y == 3, but not the same 3 in memory as the 3 literal in the first line */
```

Variable y is assigned a copy of the value of variable x during declaration. It does not point to x, so any change that occurs to either variable does not reflect the other.

```
y = 5;
int z = x = y; /* x, y and z all have value 5 now */
```

multiple assignment is possible in the same statement, and is evaluated as such: z = (x = (y)), y returns 5 and is assigned to x, (x=5) returns 5 and is assigned to z. See Section 4.2 for more on operator associativity.

3.6 Scope

Local variables are scoped in {} enclosing blocks and can only be accessed within this context. Top-level variables that are defined outside of a {} block are considered global variables and can be accessed from any defined scope.

e.g.

```
if (expr) {
    int x = 2;
}
x = 3; /* error, out of scope */

if (expr){
    int x = 2;
    if (expr) {
        x = 3; /* OK */
    }
    x = 4; /* OK */
}
x /* x == 4 */
```

4. Expressions

Expressions consist of a combination of variables, operators and other expressions that return a single value.

4.1 Expression Operators

Operator	**Description**	**Examples**
+	Arithmetic Addition, Binary Operator between two ints	x + y : between 2 vars x + 1 : between var and literal 1 + 2 : between 2 literals (1+2) + (3+4) : between 2 expressions that return type int
-	Arithmetic Subtraction, Binary Operator between two ints	x - y : between 2 vars x - 1 : between var and literal 1 - 2 : between 2 literals (1+2) - (3+4) : between 2 expressions that return type int
/	Arithmetic Division, Binary Operator between two ints	x / y : between 2 vars x / 1 : between var and literal 1 / 2 : between 2 literals (1+2) / (3+4) : between 2 expressions that return type int

*	Arithmetic Multiplication, Binary Operator between two ints	x * y : between 2 vars x * 1 : between var and literal 1 * 2 : between 2 literals (1+2) * (3+4) : between 2 expressions that return type int
%	Modulus	x % y: between 2 vars x % 2: between var and literal 1 % 2: between 2 literals (1+2) % (3+4) : between 2 expressions that return type int
++x	Pre-Increment Operator on variables of int type	int x = 3; x++; /* x == 4 */ int y = ++x; /* y == 5, x == 5 */
x++	Post-Increment Operator on variable of int type	int x = 3; ++x; /* x == 4 */ int y = x++; /* y == 4, x == 5 */
>	greater than	int x = 2; x > 1 /* true */ x > 4 /* false */
>=	greater than equal to	int x = 2; x >= 2 /* true */ x >= 4 /* false */
<	less than	int x = 2; x < 1 /* false */ x < 4 /* true */
<=	less than equal to	int x = 2; x <= 2 /* true */ x <= 4 /* true */

&&	boolean AND	bool b1 = true;
		bool b2 = false;
		
		(b1 && b2) /* false */
		(true && true) /* true */
		(true && false) /* false */
		(false && false) /* false */
		(false && true) /* false */

||	boolean OR	bool b1 = true; bool b2 = false;

		(b1 || b2) /* false */ (true || true) /* true */ (true || false) /* false */ (false || false) /* false */ (false || true) /* false */

!	boolean NOT	bool b1 = true;
		bool b2 = false;
		
		!b1 /* false */
		!b2 /* true */

==	Equals	int x = 2;
		x == 4 /* false */
		true == false /* false */
		int y = 2;
		x == y /* true */
		
		Primitive types compare values, lists compare length and element-element comparison

!=	Not Equals	int x = 2;
		x != 4 /* true */
		true != false /* true */
		int y = 2;
		x != y /* false */

=	Assignment	int x = 2;
		int y = x;
		x = 4;

4.2 Operator Precedence and Associativity

To resolve ambiguity of expressions, we set the following operator precedence and associativity of operators (in order of increasing precedence). Operators on the same level have the same precedence and are evaluated in order according to the defined associativity.

Operator(s)	**Associativity**
=	right-to-left
||	left-to-right

&&	left-to-right
== !=	left-to-right
< <= > >=	left-to-right
+ -	left-to-right
* / %	left-to-right
++ -- !	right-to-left
[:] list subscript () function call	left-to-right

e.g.
int1 + int2 * int3 is evaluated as int1 + (int2 * int3)
expr1 || expr2 && expr3 is evaluated as expr1 || (expr2 && expr3)
!expr1 || !expr2 is evaluated as (!expr) || (!expr2)
!expr1 >= expr2 || expr3 && expr4 < expr5 is evaluated as
((!expr) >= expr2) || (expr && (expr4 < expr5))

4.3 Parenthesis

Parenthesis can be explicitly added to an expression to provide explicit evaluation precedence. The deeper the () nesting, the higher the precedence.

e.g.

```
40 / 2 + 3 * 4;
default evaluation is (40 / 2) + (3 * 4) = 32
alternative with parens: 40 / (( 2 + 3 ) * 4) = 2

expr1 || expr2 && expr3;
default evaluation is expr1 || (expr2 && expr3)
alternative with parens: (expr1 || expr2) && (expr3)
```

5. Statements

The bodies of programs in AP++ are made of statements, which occur in the form of simple or compound statements.

5.1 Simple Statements

Simple statements do not contain other statements and are terminated by a semicolon. E.g.

```
int x = 1;   /* declaration + assignment */
x = 2;       /* assignment */
1+2;         /* in-line expression, which is ignored */
foo(x);      /* function call */
x = foo(x);  /* assignment to value of function call */
```

5.2 Compound Statements

Compound statements contain one or more statements for execution contained in {} blocks and have two general purposes: conditional control and iteration.

Conditional Control: *if-elseif-else* statements

For conditional control, AP++ provides *if-elseif-else* statements for executing statements upon meeting boolean conditions. both elseif and else blocks are optional.

e.g.

```
if (expression) {
    /* statements here */
}

if (expression) {
    /* statements here */
} elseif (expression2) {
    /* statements here */
}

if (expression) {
    /* statements here */
} elseif (expression2) {
    /* statements here */
} else {
    /* statements here */
}
```

Iteration: *while* loops

For iteration, AP++ provides while statements for executing statements in a loop as long as the provided expression is *true*.

e.g.

```
while (expression) {
    /* statements here */
}

int x = 4;
```

```
int fac = 1;
while (x > 0) {
    fac *= x--;
}
fac /* evaluates to 4! = 24  */
```

while loops can be terminated early with the *break* statement.

e.g.

```
while (expr) {
    if (expr) {
       break;
    }
    /* statements */
}
```

while loops can continue to the next iteration with the *continue* statement.

e.g.

```
while (expr) {
    if (expr) {
       continue;
    }
    /* statement */
}
```

6. Functions

Functions are a group of statements that can be executed via the function identifier and an optional list of function parameters as inputs. The last function in the program file must define the main function responsible for executing the program.

6.1 Return Types

Functions may return any valid type supported in the language, i.e. *int bool int[] and bool[]*. For these types of functions, the last statement must *return* the value of type expected by the function declaration. Functions may also be defined as *void* if they do not expect to return any value.

6.2 Declaration

Functions are declared in the following format:

```
(type|void) identifier((params)) {
       /* statements */
```

```
      (return type;)
}

e.g.
int foo() {
    return 3;
}

bool foo(int x) {
    return x > 5;
}

void foo(bool b) {
    /* statements */
}
```

6.3 Parameters

Function parameters are comma delimited and precede by a type identifier. Primitive type parameters are passed by value. That is, the contents of the variable passed into the function call are copied and then passed to the function and assigned to the parameter variable. Changing the value of these parameter variables will not change the value of the original variable. For more complex types such as lists, the variable will still refer to the original value in memory so performing operations on that variable will be reflected in all variables that point to it. (See Section 7.4)

e.g.

```
void square(int x) {
    x = x * x;
}

int z = 3;
foo(z) /* z == 3 after this line */
```

7. Lists

Lists in *AP++* are heavily inspired by Python's list syntax and API.

7.1 Declaration

```
int[] x;                  /* with no value assigned */
int[] x = [1, 2, 3];      /* with initializer */
```

7.2 API Methods

Method Name	**Description**
int length()	Returns the number of elements in the list
T pop()	Removes the last element in the list and returns it
T shift()	Removes the first element in the list and returns it
push(T x)	Pushes element x to the end of the list
insert(int i, T x)	Inserts element x at index i in the list
T remove(int i)	Removes element at index i and returns it
reverse()	Reverses the elements in the list in place
clear(T[] x)	Clears all elements in the list in place

7.3 List Slicing Syntax

With Python-style list slicing syntax, sublists with specified range can be returned. It has the following format: list_identifier[(int):(int)] with the optional ints as starting/ending index span to return from the original list. It is important to note that the ending index is not inclusive.

e.g.
```
int x[] = [3, 4, 5, 6, 7, 8];
x[1:4] /* evaluates to [4, 5, 6] */
x[3:]  /* evaluates to [6, 7, 8] */
x[:3]  /* evaluates to [3, 4, 5] */
x[:]   /* returns copy of x, evaluates to [3, 4, 5, 6, 7, 8]
x[2:100] /* same as x[2:], evaluates to [5, 6, 7, 8] */
```

Negative indices can be used to indicate the index from the end.

e.g.
```
x[-1] /* evaluates to 8 */
x[1:-1] /* evaluates to [4, 5, 6, 7] */
x[:-1] /* evaluates to [3, 4, 5, 6, 7] */
x[-100:3] /* evaluates to [3, 4, 5], same as [:3]
```

Start/end indices that overlap each other will return empty list

e.g.
```
x[1:-1] /* evaluates to [] */
```

7.4 Local and Function Parameter Variable Assignment

Unlike primitive types, assignment of list types point to the same allocated list in memory. Performing operations on a list will be reflected in all variables that have assigned themselves to that list. Reassigning the variable, however, will not affect the original allocated list's contents, but rather re-point the variable to another allocated list in memory.

```
int[] x = [0, 1, 2];
int[] y = x;
y.pop(); /* evaluates to 2, x and y are now both [0, 1] */
x[-1] /* evaluates to 1 */
int[] z = [3, 4, 5];
z = x;
x /* unchanged, still evaluates to [0, 1, 2] */
z /* evaluates to [3, 4, 5] */
z.pop() /* evaluates to 5, z is now [3, 4] */
x /* evaluates to [3, 4], same as z
```

```
void foo(int[] x) {
    x.pop();
}

int[] x = [0, 1, 2];
foo(x);
x[0]; /* evaluates to 1 */
```

7.4 Equality

The == operator between lists first performs a size comparison and then performs element-to-element equality check by value.

e.g.

```
int x[] = [0, 2, 3];
int y[] = [0, 2];

x == y /* evaluates to false */

int z[] = [0, 2, 3];
z == x /* evaluates to true */
```

8. Dicts

Time-permitting, AP++ will also support Python-style dictionary data structure.

9. Program Examples

Euclidean Algorithm (GCD)

```
int gcd(int x, int y) {
    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}

void main() {
    gcd(10, 15);
}
```

Merge Sort

```
// merges two sorted sublists of arr[] (arr[0..m], arr[m+1..r]) in-place.
void merge(int[] arr, int l, int m, int r) {
    // temp lists for l and r sides
    int[] L = arr[0:m];
    int[] R = arr[m+1:r];

    // merge the temp lists back into arr[l..r]
    int i = 0;      // init index of 1st sublist
    int j = 0;      // init index of 2nd sublist
    int k = l;      // init index of merged sublist

    while (k < r) {
        if (j >= r || (i < m && L[i] <= R[j])) {
            arr[k] = L[i];
            i++;
        } else if (i >= m || (j < r && L[i] > R[j])) {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
}

void mergeSortImpl(int[] list, l, r) {
    if (l >= r) {
        return;
    }
    int m = (l + (r-1)) / 2;
    mergeSort(list, l, m);
    mergeSort(list, m+1, r);
    merge(list, l, m, r);
}
```

```
void main() {
    int[] l = [5, 2, 0, 3, 6];
    mergeSort(l, 0, 4) /* l => [0, 2, 3, 5, 6];
}
```