Bear 언어 소개

목차

1. 개요

- 본 문서는 프로그래밍 언어 Bear 의 문법을 정의한다. Bear 언어는 C 언어를 기반으로 하는 절차지향적 언어로, C 언어와는 달리 동적으로 크기가 변하는 컨테이너를 기본타입으로 제공한다.

2. 문법 정의

1) Type

1-1) int

* 저장되는 값 : integer literal
* 값의 표현방식 : 2 의 보수
* 메모리 공간의 크기 : 4bytes

1-2) list

* 저장되는 값 : integer literal, array, list 중 0 개 이상
* 값의 표현방식 : 2 의 보수
* 메모리 공간의 크기 : 4*M bytes

1-3) C 언어 type과 차이점

- int 는 C 언어 제공되는 type 과 동일하며, 그에 적용되는 연산자 및 연산의 결과도 동일하다.
- list type 이 추가 제공되며, 이에 대해서는 '6) list(향상점)'에서 자세히 다루도록 한다.

2) Control Statement

2-1) if(조건문)

* 조건문 : true/false 의 진리값
  (단, int 에서 0 은 false 로, 그 외의 값은 true 로 간주한다.)

* 제약조건 :

- if 문은 바로 다음 1 개의 문장에만 영향을 준다. 즉, if 의 조건문이 true 이면 if 문 바로 뒤의 문장만 수행된다.
- if 문은 독립적으로 존재할 수 있다.(else 문이 없어도 존재할 수 있다.)

* 동작메커니즘
[1] : 조건문의 true/false 를 판단한다.
[2-t] : [1]의 결과가 true 이면 if 문에 속한 문장을 수행한다.
[2-f] : [1]의 결과가 false 이면 if 문에 속한 문장을 수행하지않는다. else 가 있다면, else 문에 속한 문장을 수행한다.

2-2) else

* 제약조건
- else 문은 바로 다음 1 개의 문장에만 영향을 준다.
- else 문은 독립적으로 존재할 수 없다.(반드시 else 문 전에 if 문이 존재해야 한다.)
* 동작메커니즘
[1-1] if 의 조건문이 true 이면 else 문에 속한 문장을 수행하지 않는다.
[1-2] if 의 조건문이 false 이면 els 문에 속한 문장을 수행한다.

2-3) while(조건문)

* 조건문 : true/false 의 진리값
(단, int 에서 0 은 false 로, 그 외의 값은 true 로 간주한다.)
* 제약조건
- while 문은 바로 다음 1 개의 문장에만 영향을 준다. 즉, while 의 조건문이 true 이면 while 문 바로 뒤에 한개의 문장만 수행된다.
- while 문은 독립적으로 존재할 수 있다.
* 동작메커니즘
[1] 조건문의 true/false 를 판단한다.
[2-1] [1]의 결과가 false 이면 while 문에 속한 문장을 수행하지않고 while 문을 끝낸다.
[2-2] [1]의 결과가 true 이면 while 문에 속한 문장을 수행한 후 [1]로 돌아간다.

2-4) C 언어 Control Statement과 차이점

Control Statement 는 기존 C 언어에서 제공되는 if-else, while 과 동작메커니즘이 동일하다.

3) Subprograms

3-1) Function, Procedure

Function 은 특정한 동작을 수행하는 일련의 명령을 말하며, Procedure 는 function 의 한 종류로, 리턴 값이 없는 function 을 지칭한다. function 의 형태는 다음과 같다.

```
return_type function_name(parameter list)
{
        // function definition
        return 'returnvalue' ;
}
```

3-2) C 언어 Subprograms과의 차이점

- list 를 통해 서로 다른 타입의 여러 개의 값을 return 할 수 있다.
- list 를 통해 서로 다른 타입의 여러 개의 값을 parameter 로 한번에 전달 받을 수 있다.

4) Operator

4-1) Arithmetic Operator

* 종류 : +, -, *, /
* 피연산자 : int type 의 변수 또는 integer literal 2 개
* 연산의 결과 : integer literal
* 형태
  infix 방식으로 연산을 하며, 연산의 일반적인 형태는 다음과 같다.
  - (피연산자 1) (연산자) (피연산자 2)

4-2) Assignment Operator

* 종류 : =

* 피연산자
  - lvalue : int type 의 변수 또는 list name
  - rvalue :
    [1] lvalue 가 int type 의 변수인 경우, int type 의 변수 또는 integer literal
    [2] lvalue 가 list name 인 경우, list name
* 연산의 결과 : lvalue(int type 의 변수 또는 list)

* C 언어 Assignment Operator 과 차이점
- list 에 대해 Assignment Operator 가 적용되며, deep copy 로 수행된다. 즉, list 에 대해 Assignment Operator 적용 시, source 와 destination list 는 각각 별개의 메모리 공간을 차지한다.

4-3) Relational Operator

* 종류 : <=, >=, <, >, ==, !=
* 피연산자 : int type 의 변수 또는 integer literal 2 개
* 연산의 결과 : true/false 의 진리값
  (단, int 에서 0 은 false 로, 그 외의 값은 true 로 간주한다.)

* C 언어 Relational Operator 과 차이점
- Relational Operator 는 기존 C 언어에서 int type 에 대해 제공되는 연산자의 동작메커니즘 및 결과가 동일하다.

5) Input/Output

5-1) Input

* 방식 : standard input(키보드 입력)
* 함수 : read(피연산자)
* 피연산자 : int type 의 변수(명)
* 제약사항
  - read()함수는 1 개의 피연산자를 가진다.

5-2) Output

* 방식 : standard output(모니터 출력)
* 함수 : write(피연산자)
* 피연산자 : int type 의 변수(명) 또는 list(명)

* 제약사항
  - write()함수는 1 개의 피연산자를 가진다.

6) list(향상점)

기존 C 언어와 차이점은 'list'라는 새로운 type 이 추가되었다는 점이며, list 의 특징은 다음과 같다.

- 서로 다른 타입의 값을 저장할 수 있다.
- run-time 도중에 길이가 바뀔 수 있다.
- list type 간의 assignment operator 가 지원된다.
- assignment 연산은 deep copy 로 수행되며, assignment operator 를 적용 후 source list 와 destination list 는 독립적으로 존재한다.
- list 에서 특정 index 에 저장된 값을 제거할 수 있다. 제거는 다음과 같이 수행한다.
  ^list_name[index]; // list 의 요소가 제거된다.
  ^list_name[index] = 5; // syntax error

- list 를 함수 인자로 받을 수 있다. 인자로 넘겨준 list 는 함수에 의해 값이 바뀔 수 있다. (CBR 방식)
- list 를 함수에서 return 할 수 있다. 기존 C 언어에서 두 개 이상의 값을 return 하는 것이 불가능했는데 list 에 담아서 return 하면 두 개 이상의 값을 return 하는 것이 가능하다.

3. Bear 언어 프로그램 예제

1) declaration

example 1) list a;
example 2) list a = {1,2,3};
example 3) list a = {1,2,{1,2,3}};
example 4) list b = a; // a is another list

2) assignment operator

example 1)

```
== source code ==
list a = {1,2,3};
list b;
b = a;
write(b);
-----------------
==== result ====
1 2 3
-----------------
```

example 2)

```
== source code ==
list a = {1,2,3};
list b;
b = a;
b[0] = 2; // list 'a' does not change.
write(a);
write(b);
-----------------
==== result ====
1 2 3
2 2 3
-----------------
```

3) input & output

example 1)

```
== source code ==
list a;
a[3] = 1;
a[5] = 2;
a[1] = 7;
```

```
write(a); // print in index order
-----------------
==== result ====
7 1 2
-----------------
```

4) function

example 1)

```
== source code ==
list func(void)
{
        int v1 = 11, v2 = 22;
        list a = {v1,v2};
        return a;
}

list b = func();
write(b); // print in index order
-----------------
==== result ====
11 22
-----------------
```

example 2)

```
== source code ==
void func(list a)
{
        a[0] = a[0]+1;
        return a;
}

list a = {11,22};
func();
write(a); // print in index order
-----------------
```

==== result ====

12 22
