**H++ Language 최종 보고서**

**작성자: 이현호, 이호욱**

**작성일자: 2016.12.20**

1. **Synopsis**
   1. Language

H++는 C언어를 기반으로 하여 기존 C언어의 문법으로는 반복문(for, while), 조건 분기문(if then else)을, 자료형으로는 int, float을 지원한다. 기존 C언어에 존재하던 switch 분기문과 struct, enum, char 자료형은 지원하지 않고, char 자료형 대신 string 자료형을 사용하여 문자열을 나타내고 포인터와 배열 중 포인터를 제외하고 배열만 지원한다. Comment는 “//”로 한줄 씩 처리한다.

* 1. Special Features
     1. 함수 overloading
        + C 는 함수 overloading을 지원하지 않지만 H++언어는 함수 overloading을 지원하여 같은 이름의 함수더라도 파라미터의 개수와 type에 따라 각각의 함수를 구별할 수 있다.
     2. for loop 2가지 지원
        + C 에서는 for loop의 parameter로 정수형 자료형이 오지만 H++언어에서는 for(type var; Exp; Exp)의 형태와 for(type var : array )의 형태도 지원한다. for(type var; Exp; Exp)는 C언어에서와 같고 for(type var : array)는 var에 array[0]부터 끝까지 차례대로 저장한다.
     3. 축약 함수
        + python의 lambda 함수를 추가하여 var = lambda( 인자 : 표현식 )을 사용하여 간단한 함수를 바로 만들어낼 수 있다.
     4. 자료형
        + Character 자료형을 사용하여 문자와 문자열을 표현하는 C와 다르게 String 자료형을 사용하고 Character 자료형은 제외하였다.
        + python에서 사용하는 것과 같은 tuple type을 지원하기 때문에 함수의 return을 (var, var)과 같이 할 수 있다. 따라서 함수를 호출할 때 a,b,c = func(); 같은 형식으로 사용하고 반환되는 튜플을 둘 이상의 변수에 각각 할당할 수 있다.
     5. Yield 지원
        + C에는 없는 좋은 기능 Yield를 지원한다. Python에서 yield문을 가져와서 함수를 잠시 중단하고 중간 결과값을 반환한다. Next()를 호출하면 중단된 지점 뒤부터 함수를 계속 실행한다.

1. **Lexical Issue**
   1. Data Type
      1. Scalar type
         * integer, float, boolean
      2. Vector type
         * List, String, Scalar type token 뒤에 [] 형태(Array)
         * character 자료형을 지원하지 않고 모두 String으로 사용한다.
         * 다차원 정의도 가능
      3. Tuple type
         * 괄호 안에 Scalar type이 있는 형태 (Scalar type, Scalar type, …)
         * 내부의 값은 변경할 수 없다.
   2. Function
      1. 모든 함수는 하나의 Data Type을 반환, Tuple타입을 사용하면 호출자에서 둘 이상의 변수에 return 값을 담을 수 있다.
      2. 모든 함수는 파라미터로 모든 Data Type을 가질 수 있음
   3. Branch
      1. IF – ELSEIF-ELSE 구문을 사용한다.
   4. Iteration
      1. For, While 구문을 사용하고 For 구문의 경우 for( ; ; )의 형태와 for(var : array)의 형태 두가지를 지원한다.
   5. Comments
      1. //
         * // 뒤부터 개행 문자를 만날 때까지 컴파일러는 해당 문자 무시
2. **Grammar**
   1. Symbol
      1. *[* x *]* : x가 0번 혹은 1번 나올 수 있음
      2. x\*or *{* x *}*: x가 0번 이상 나올 수 있음
      3. x+ : x가 1번 이상 나올 수 있음
      4. x | y: x 혹은 y가 1번 나올 수 있음
   2. Concrete Syntax

|  |
| --- |
| Program → FuncDecl\* MainFunc  MainFunc → Type main **{** Statement\***}**  Lambda→ **(**VarDecl **:** Exp**)**  VarDecl → Type id *{* **,** id *}***;**  FuncDecl → Type id **(** ParameterList **) {** FuncElements\* **}**  ParameterList → ε|Parameter *{***,** Parameter*}*  Parameter → Type id  FuncElements → VarDecl | Statement  Statement → **{** Statement\* **}**  → Exp;  → If\_Statement  → While\_Statement  → For\_Statement  → IO\_Statement  → **yield** Exp;  → **return** Exp**;**  → **;**  If\_Statement → **if** **(** Exp **)** Statement *{* **elseif (**Exp**)** Statement *}* *[* **else** Statement *]*  While\_Statement → **while** **(** Exp **)** Statement  For\_Statement → **for** **(** ForInit **;** Exp **;** Exp **; )** Statement  → **for** **(** id **:** id **)** Statement  ForInit → Exp**;** | **;**  IO\_Statement → **printf (**Exp**)**  → **scanf(**Exp**)**  Exp → Exp1 *[* AssginOp Exp1 *]*  Exp1 → Exp2 *[* Exp1Rest *]*  Exp1Rest → *[* **?** Exp **:** Exp1 *]*  Exp2 → Exp3 *[* Exp2Rest *]*  Exp2Rest → *{* InfixOp Exp3 *}*  Exp3 → PrefixOp Exp3  → Primary Selector\*  Primary → ( Exp )  → id  → **this**  → Literal  → **new** id **(** ExpList **)**  Selector → **.**id *[* **(** ExpList **)** *]*  → **.**VectorOp  → **.**YieldOp  → **[** Exp **]** ExpList → ε|Exp *{* **,** Exp *}*  Type → ScalarType | VectorType | TupleType | id ScalarType → **int** | **boolean** | **float**  TupleType→ **(**ScalarType *{***,** ScalarType*}* **)**  VectorType → ScalarType **[** IntegerLiteral **]***{* **[** IntegerLiteral **]** *}*  → **List <** Type **>**  → **String**  VectorOp → **add(** Exp **)** | **length()** | **reverse()** | **init()** | **end()**  YieldOp→ **next()**  PrefixOp → !              → ++              → --  → ~  AssignOp → =  → +=  → -=  → \*=  → /=  → %=  → ^=  InfixOp → +              → -              → \*              → /              → %              → ^              → <              → <=              → >              → >=              → !=              → ==              → &&              → ||  Literal → IntegerLiteral | StringLiteral | BooleanLiteral  IntegerLiteral    → 0-9  FloatLiteral → IntegerLiteral**.**IntegerLiteral  StringLiteral → *{* a-zA-Z | \_*}*  BooleanLiteral → true|false  id → StringLiteral *{* IntegerLiteral *}* |

* 1. Abstract Syntax

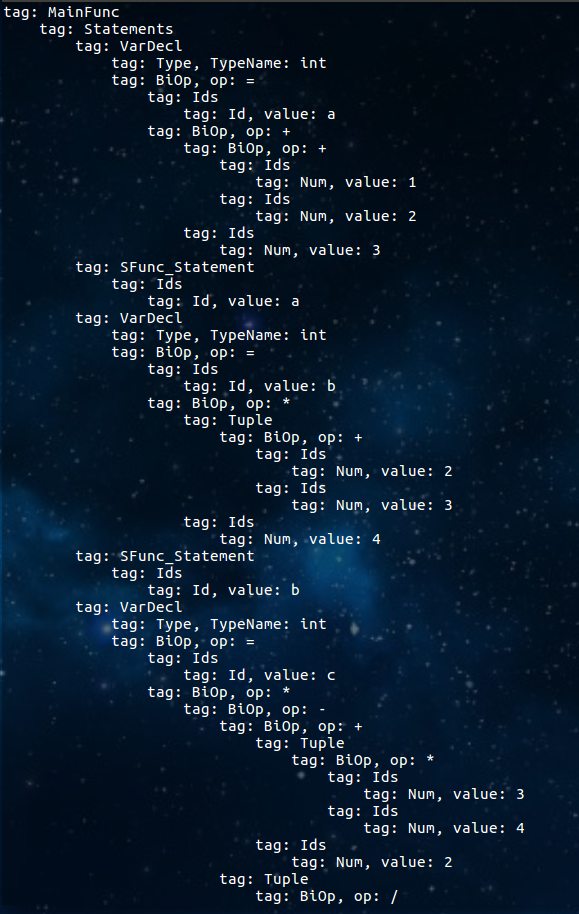
|  |
| --- |
| Program → FuncDecl\* MainFunc  MainFunc → Type main ‘{‘VarDecl\* Statement\* ‘}’  Lambda→ ‘(‘ VarDecl ‘:’ Exp ‘)’  VarDecl → Type ID {‘,’ ID}\*  FuncDecl → Type ID ‘(‘ParameterList ‘)’ ‘{‘FuncElements\* ‘}’  ParameterList → Parameter {‘,’ Parameter }\*  Parameter → Type ID  FuncElements → VarDecl | Statement  Statement → ‘{‘ Statement\* ‘}’  | Exp’;’  | If\_Statement  | While\_Statement  | For\_Statement  | IO\_Statement  | ‘yield’ Exp;  | ‘return’ Exp’;’  | ‘;’  If\_Statement → ‘if’ ‘(‘ Exp ‘)’ Statement {‘elseif’‘(‘Exp’)’Statement }\*  *[* ‘else’ Statement *]*  While\_Statement → ‘while’ ‘(‘ Exp ‘)’ Statement  For\_Statement → ‘for’ ‘(‘ ForInit ‘;’ Exp ‘;’ Exp‘)’ Statement  | ‘for’ ‘(‘ ID ‘:’ ID ‘)’ Statement  ForInit → Exp’;’ | ‘;’  IO\_Statement → ‘printf’‘(‘Exp’)’  | ‘scanf’’(‘Exp’)’  Exp → Exp AssginOp Exp  | Exp ‘?’ Exp ‘:’ Exp  | Exp InfixOp Exp  | PrefixOp Exp3  | Primary Selector\*  Primary → ‘(‘ Exp ‘)’  | ID  | ‘this’  | Literal  | ‘new’ ID ‘(‘Exp {‘,’ Exp }\* ‘)’  Selector → ‘.’ID *[* ‘(‘Exp { ‘,’ Exp }\* ‘)’ *]*  | **.**VectorOp  | ‘[‘ Exp ‘]’  | ‘.’YieldOp  Type → ScalarType | VectorType | TupleType | id ScalarType → ‘int’ | ‘boolean’ | ‘float’  TupleType→ ‘(‘ScalarType {‘,’ ScalarType}\* ‘)’  VectorType → ScalarType ‘[‘IntegerLiteral ‘]’ {‘[‘IntegerLiteral ‘]’}\*  | ‘List’‘<’ Type ‘>’  | ‘String’  VectorOp → ‘add’‘(’ Exp ‘)’ | ‘length’‘(’‘)’ | ‘reverse’‘(’‘)’ | ‘init’‘(’‘)’  | ‘end’ ‘(’ ‘)’ | ‘sort’‘(’‘)’  YieldOp→‘next’‘(’‘)’  PrefixOp → ! | ++ | -- | ~  AssignOp → = | += | -= | \*= | /= | %= | ^=  InfixOp → + | - | \* | / | % | ^ | < | <= | > | >= | != | == | && | ||  Literal → IntegerLiteral | StringLiteral | BooleanLiteral  IntegerLiteral    → 0-9  FloatLiteral → IntegerLiteral’.’IntegerLiteral  StringLiteral → {a-zA-Z | \_}\*  BooleanLiteral → true | false  ID → StringLiteral {IntegerLiteral}\* |

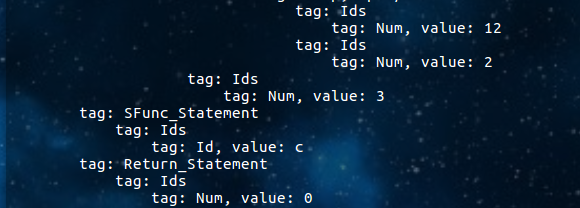
1. **Sample Program**

|  |
| --- |
| int main(){  int a = 1 + 2 + 3;  printf(a);  int b = (2+3) \* 4;  printf(b);  int c = (3 \* 4) + 2 - (12 / 2) \* 3;  printf(c);  return 0;  } |

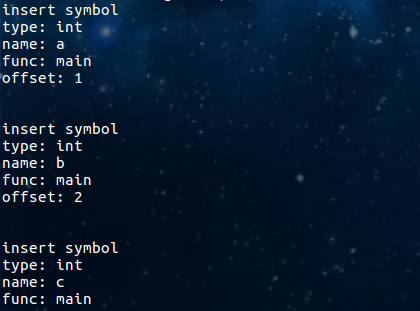
**결과**

**1. Abstract Syntax Tree**

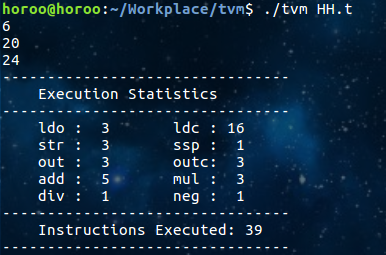




**2. Symbol Table**



**3. Execute**



1. **결론**

기본적으로 문법 설계에 미숙한 점이 많아 AST를 만들 때 Shift/Reduce, Reduce/Reduce를 없애는 과정에 시간이 오래 걸렸다.

AST를 구현한 뒤 Symbol Table을 만드는 부분에서도 오랜 시간이 걸렸고 한 단계씩 진행할 때마다 이전 단계에서 생겨난 에러가 나타났고 시간 역시 촉박하여 당장 고쳐야 되는 부분만 수정하다 보니 다시 에러가 우후죽순으로 생겨나는 악순환이 발생하였다.

AST는 Special Feature에 맞춰 생성할 수 있었지만 Code Generation은 간단한 사칙연산에 대한 결과를 생성하도록 만들었다.

텀 프로젝트를 진행 하면서 컴파일러의 동작 과정에 대해 깊이 생각해 볼 수 있었다.

미완인 부분들은 좀 더 시간을 가지고 차근차근 생각해 보며 완성해 볼 예정이다.

1. **Reference**
2. <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>
3. [https://docs.python.org/2/reference/index.html](https://docs.python.org/2/reference/index.html%20)
4. [https://docs.python.org/2/reference/datamodel.html#objects-values-and-types](https://docs.python.org/2/reference/datamodel.html%23objects-values-and-types)
5. <http://www.cplusplus.com/doc/tutorial/functions2/>
6. [https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14.2](https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html%23jls-14.14.2)