Experiment in
Compiler Construction

# Symbol table

ONE LOVE. ONE FUTURE.

# Contents

- Overview
- Symbol table

# Symbol table

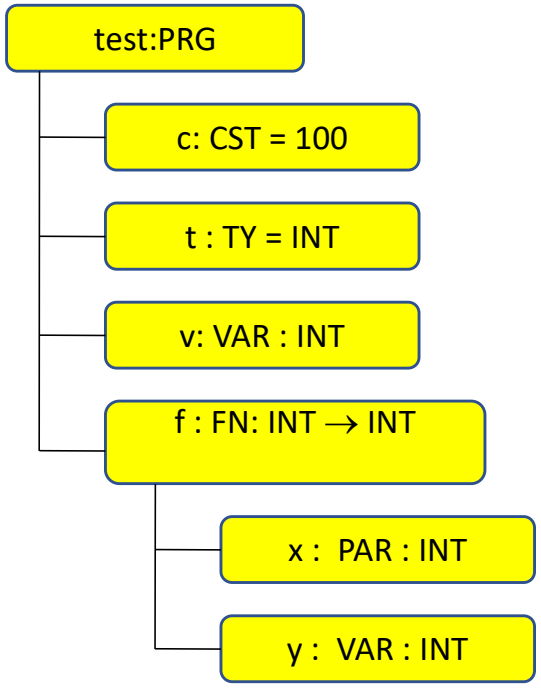
- It maintains all declarations and their attributes
  - Constants: {name, type, value}
  - Types: {name, actual type}
  - Variables: {name, type}
  - Functions: {name, parameters, return type, local declarations}
  - Procedures: {name, parameters, local declarations)
  - Parameters: {name, type, call by value/call by reference}

# Symbol table

- In a KPL compiler, the symbol table is represented as a hierarchical structure

```
PROGRAM test;
CONST c = 100;
TYPE t = Integer;
VAR v : t;
FUNCTION f(x : t) : t;
  VAR y : t;
BEGIN
  y := x + 1;
  f := y;
END;

BEGIN
  v := 1;
  Call WriteI (f(v));
END.
```



# Symbol table implementation

- Elements of the symbol table

```
// symbol table
struct SymTab_ {
  // main program
  Object* program;

  // current scope
  Scope* currentScope;

  // Global objects such as
  // WRITEI, WRITEC, WRITELN
  // READI, READC
  ObjectNode
   *globalObjectList;
};
```

```
// Scope of a block
struct Scope_ {
  // List of block's objects
  ObjectNode *objList;

  // Function, procedure or program
   that
  //block belongs to
  Object *owner;

  // Outer scope
  struct Scope_ *outer;
};
```

# Symbol table implementation

- Symbol table has `currentScope` tell current block
- Update `currentScope` whenever beginning parsing a procedure/function

```
void enterBlock(Scope* scope);
```

- Return `currentScope` to outer block whener a procedure/function has been analysed

```
void exitBlock(void);
```

- Declare a new object in current block

```
void declareObject(Object* obj);
```

# Constant and Type

```
// Type classification

enum TypeClass {
  TP_INT,
  TP_CHAR,
  TP_ARRAY
};

struct Type_ {
  enum TypeClass
   typeClass;

  // Use for type Array
  int arraySize;

  struct Type_
   *elementType;
};
```

```
// Constant

struct ConstantValue_ {
  enum TypeClass type;
  union {
    int intValue;
    char charValue;
  };
};
```

# Constant and Type

- To make type

```
Type* makeIntType(void);
Type* makeCharType(void);
Type* makeArrayType(int arraySize, Type* elementType);
Type* duplicateType(Type* type)
```

- To make constant value

```
ConstantValue* makeIntConstant(int i);
ConstantValue* makeCharConstant(char ch);
ConstantValue*
   duplicateConstantValue (ConstantValue* v);
```

# Object

```
// Object
// classification

enum ObjectKind {
  OBJ_CONSTANT,
  OBJ_VARIABLE,
  OBJ_TYPE,
  OBJ_FUNCTION,
  OBJ_PROCEDURE,
  OBJ_PARAMETER,
  OBJ_PROGRAM
};
```

```
// Objects' attributes in symbol
// table

struct Object_ {
  char name[MAX_IDENT_LEN];
  enum ObjectKind kind;
  union {
    ConstantAttributes* constAttrs;
    VariableAttributes* varAttrs;
    TypeAttributes* typeAttrs;
    FunctionAttributes* funcAttrs;
    ProcedureAttributes* procAttrs;
    ProgramAttributes* progAttrs;
    ParameterAttributes* paramAttrs;
  };
};
```

# Object – Object's attributes

```
struct ConstantAttributes_ {
  ConstantValue* value;
};
struct VariableAttributes_ {
  Type *type;
  // Scope of variable (for code generation)
  struct Scope_ *scope;
};
struct TypeAttributes_ {
  Type *actualType;
};
struct ParameterAttributes_ {
  // Call by value or call by reference
  enum ParamKind kind;
  Type* type;
  struct Object_ *function;
};
```

# Object – Object's attributes

```
struct ProcedureAttributes_ {
  struct ObjectNode_ *paramList;
  struct Scope_* scope;
};

struct FunctionAttributes_ {
  struct ObjectNode_ *paramList;
  Type* returnType;
  struct Scope_ *scope;
};

struct ProgramAttributes_ {
  struct Scope_ *scope;
};

// Note: parameter objects are declared in list of parameters
  (paramList) as well as in list of objects declared inside current
  block (scope->objList)
```

# Object

- Create a constant object

```
Object* createConstantObject(char *name);
```

- Create a type object

```
Object* createTypeObject(char *name);
```

- Create a variable object

```
Object* createVariableObject(char *name);
```

- Create a parameter object

```
Object* createParameterObject(char *name
                              enum ParamKind kind;
                              Object* owner;);
```

# Object

- Create a function object

```
Object* createFunctionObject(char *name);
```

- Create a procedure object

```
Object* createProcedureObject(char *name);
```

- Create a program object

```
Object* createProgramObject(char *name);
```

# Free the memory

- Free a type

```
void freeType(Type* type);
```

- Free an object

```
void freeObject(Object* obj)
```

- Free a list of object

```
void freeObjectList(ObjectNode* objList)
void freeReferenceList(ObjectNode* objList)
```

- Free a block

```
void freeScope(Scope* scope)
```

# Debugging

- Display type's information
  - `void printType(Type* type);`
- Display object's information
  - `void printObject(Object* obj, int indent)`
- Display object list's information
  - `void printObjectList(ObjectNode* objList, int indent)`
- Display block's information

  `void printScope(Scope* scope, int indent)`

# Implement symbol table for KPL

- Initialize and Clean symbol table
- Constant declaration
- Type declaration
- Variable declaration
- Function/Procedure declaration
- Parameter declaration

# Initialize & Clean a symbol table

```
int compile(char *fileName) {
  …
  // Initialize a symbol table
  initSymTab();
  // Compile the program
  compileProgram();
  // Display result for checking
  printObject(symtab->program,0);
  // Clean symbol table
  cleanSymTab();
  …
}
```

# Initialize program

- The program object is initialized by

```
void compileProgram(void);
```

- After program initialization, we enter the outermost block by **`enterBlock()`**
- When program is completely analyzed, we exit by **`exitBlock()`**

# Constant declaration

- Constant objects are created and declared inside the function `compileBlock()`
- During analysing process, constants' values are filled by

```
ConstantValue* compileConstant(void)
```

  - *In case a constant's value is identifier constant, like **const b=a**;refer to symbol table to find actual value.*
- When a constant has been analysed, he has to be declared in current block by function `declareObject`

# User-defined type declaration

- Type objects are created and declared inside the function `compileBlock2()`
- Actual type is learned during the analysing by function `Type* compileType(void)`
  - If we meet identifier type, refer to symbol table to find actual type
- When a user-defined type has been analysed, he has to be declared in current block by function `declareObject`

# Variable declaration

- Variable objects are created and declared inside function

```
compileBlock3()
```

- Type of a variable is filled when analysing type by using function

```
Type* compileType(void)
```

- For later code generation, one of variable object's attributes should be the current scope.
- When a variable object is analysed, he has to be declared in current block by function `declareObject`

# Function declaration

- Function objects are created and declared in function `compileFuncDecl()`
- Attributes of a function object need to be filled include:
  - List of parameters, in function `compileParams`
  - Return type, in function `compileType`
  - Function's scope
- Note:The function object has to be declared in current block

  Update function scope as currentScope before deal with function local object.

# Procedure declaration

- Function objects are created and declared in function

  **`compileProcDecl()`**
- Attributes of a function object need to be filled include:
- List of parameters, in function `compileParams`
- Note:The function object has to be declared in current block
- Update function scope as currentScope before deal with function local object.

# Parameter declaration

- Parameter objects are created and declared in function `compileParam()`
- Parameter objects' attributes:
  - Data type of parameter: a basic type
  - Kind of parameter: Call by value (PARAM_VALUE) or call by reference (PARAM_REFERENCE)
- Note: parameter objects should be declared in both
  - Current function's list of parameter (`paramList`)
  - Current function's list of local objects (`objectList`).

# Assignments

- Review the parser with changes
- Complete the TODO-marked functions to perform object registration tasks
- Test with sample examples

# Compare with project Symtab0: How to create a program object

```
 obj =
createProgramObject("PRG");
enterBlock(obj->progAttrs-
>scope);
```

```
// TODO: create, enter,
and exit program block
  eat(KW_PROGRAM);
  eat(TK_IDENT);
  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_PERIOD);
}
```

# Compare with project Symtab0: How to create a constant object

```
obj = createConstantObject("c1");
  obj->constAttrs->value =
makeIntConstant(10);
  declareObject(obj);
  obj = createConstantObject("c2");
  obj->constAttrs->value =
makeCharConstant('a');
  declareObject(obj);
```

```
void compileBlock(void) {
  // TODO: create and declare constant
objects
  if (lookAhead->tokenType == KW_CONST) {
    eat(KW_CONST);
    do {
      eat(TK_IDENT);//Create a constant
      eat(SB_EQ);
      compileConstant();//Update the value
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType ==
TK_IDENT);

    compileBlock2();
  }
  else compileBlock2();
}
```

# Compare with project Symtab0: How to create a type object

```
obj =
createTypeObject("t1");
  obj->typeAttrs->actualType
=
makeArrayType(10,makeIntType
());
  declareObject(obj);
```

```
compileBlock2(void) {
  // TODO: create and declare type
objects
  if (lookAhead->tokenType == KW_TYPE) {
    eat(KW_TYPE);

    do {
      eat(TK_IDENT);
      eat(SB_EQ);
      compileType();
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType ==
TK_IDENT);

    compileBlock3();
  }
  else compileBlock3();
}
```

# Compare with project Symtab0: How to create a variable object

```
obj = createVariableObject("v1");
  obj->varAttrs->type = makeIntType();
  declareObject(obj);
obj = createVariableObject("v2");
  obj->varAttrs->type = makeArrayType(10,makeArrayType(10,makeIntType()));
```

- `declareObject(obj);`

```
void compileBlock3(void) {
  // TODO: create and declare variable objects
  if (lookAhead->tokenType == KW_VAR) {
    eat(KW_VAR);
    do {
      eat(TK_IDENT);
      eat(SB_COLON);
      compileType();
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);
    compileBlock4();
  }
  else compileBlock4();
}
```

# Duplicate functions (ConstantValue, Type)

## In function compileConstant2

```
case TK_IDENT:
    eat(TK_IDENT);
obj = lookupObject(currentToken->string);
    if ((obj != NULL) && (obj->kind == OBJ_CONSTANT))
      constValue = duplicateConstantValue(obj->constAttrs->value);
    else
      error(ERR_UNDECLARED_CONSTANT, lookAhead->lineNo, lookAhead-
>colNo);
    break;
```

## In function compileType

```
 case TK_IDENT:
    eat(TK_IDENT);
    obj = lookupObject(currentToken->string);
    if ((obj != NULL) && (obj->kind == OBJ_TYPE))
      type = duplicateType(obj->typeAttrs->actualType);
    else
      error(ERR_UNDECLARED_TYPE, lookAhead->lineNo, lookAhead->colNo);
```

# So sánh với bài 1: Tạo hàm

```
obj = createFunctionObject("f");
  obj->funcAttrs->returnType = makeIntType();
  declareObject(obj);
      enterBlock(obj->funcAttrs->scope);
     obj = createParameterObject("p1", PARAM_VALUE, symtab->currentScope->owner);
    obj->paramAttrs->type = makeIntType();
    declareObject(obj);
    obj = createParameterObject("p2", PARAM_REFERENCE, symtab->currentScope->owner);
    obj->paramAttrs->type = makeCharType();
    declareObject(obj);
    exitBlock();
```

```
void compileFuncDecl(void) {
  // TODO: create and declare a function object
  eat(KW_FUNCTION);
  eat(TK_IDENT);
  compileParams();//điền //paramlist.objlist
  eat(SB_COLON);
  returnType = compileBasicType();
  funcObj->funcAttrs->returnType = returnType;

  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_SEMICOLON);
}
```

# So sánh với bài 1: Tạo tham số

obj = createParameterObject("p1", PARAM_VALUE, symtab->currentScope-> owner);

obj->paramAttrs->type = makeIntType();

declareObject(obj);

obj = createParameterObject("p2", PARAM_REFERENCE, symtab->currentScope-> owner);

obj->paramAttrs->type = makeCharType();

declareObject(obj);

```
void compileParam(void) {
  // TODO: create and declare a parameter
  switch (lookAhead->tokenType) {
  case TK_IDENT://tham trị
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
    break;
  case KW_VAR://tham biến
    eat(KW_VAR);
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
    break;
  default:
    error(ERR_INVALID_PARAMETER, lookAhead->lineNo, lookAhead->colNo);
    break;
  }
}
```

# Add a parameter to 2 lists

```
void declareObject(Object* obj) {
  if (obj->kind == OBJ_PARAMETER) {
    Object* owner = symtab->currentScope->owner;
    switch (owner->kind) {
    case OBJ_FUNCTION:
      addObject(&(owner->funcAttrs->paramList), obj);
      break;
    case OBJ_PROCEDURE:
      addObject(&(owner->procAttrs->paramList), obj);
      break;
    default:
      break;
    }
  }

  addObject(&(symtab->currentScope->objList), obj);
}
```