ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

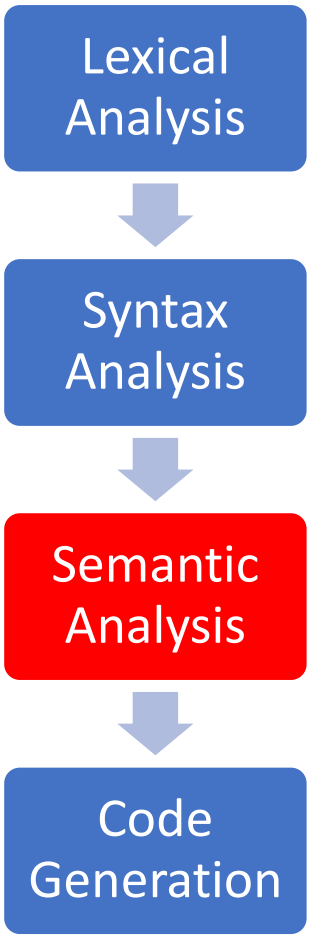
Experiment in Compiler Construction

Semantic Analysis (1)
Symbol table

Content

- Overview
- Symbol table

What is semantic analysis?



- Syntax analysis checks only grammatical correctness of a program
- There are a number of correctness that are deeper than grammar
  - Is “x” a variable or a function?
  - Is “x” declared?
  - Which declaration of “x” does a given use reference?
  - Is the assign statement “c:=a+b” type consistent?
  - …
- Semantic Analysis answers those questions and gives direction to a correct code generation.

Tasks of a semantic analyzer

- Maintaining information about identifiers
  - Constants
  - Variables
  - Types
  - Scopes (program, procedures, and functions)
- Checking semantic rules
  - Scoping rules
  - Typing rules
- Invoking code generation routines

Symbol table

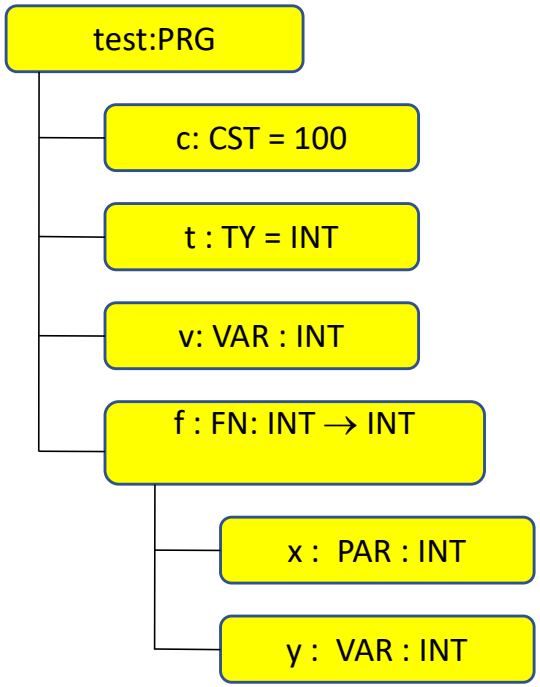
- It maintains all declarations and their attributes
  - Constants: {name, type, value}
  - Types: {name, actual type}
  - Variables: {name, type}
  - Functions: {name, parameters, return type, local declarations}
  - Procedures: {name, parameters, local declarations)
  - Parameters: {name, type, call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