

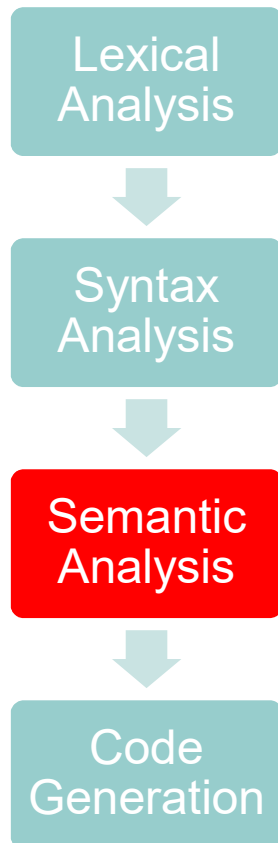
Experiment in Compiler Construction Semantic Analysis (1)

School of Information and Communication
Technology
Hanoi University of Science and
Technology

Content

- Overview
- Symbol table
- Static semantic analysis

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

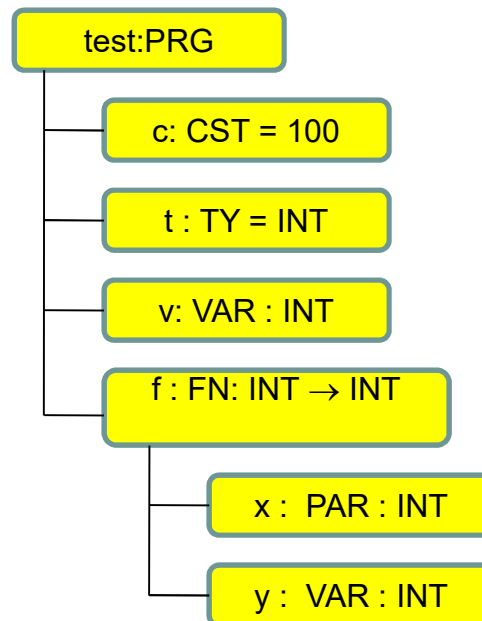
Symbol table

- 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;  
    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 whenever 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)
```

Semantic analyzer - organization

#	File name	Task
1	makefile	Project
2	symtab.c, symtab.h	Symbol table implementation
3	debug.c, debug.h	Debugging
4	main.c	Main program

Assignment 1

- Implement symbol table: Complete *TODO* function in *syntab.c*

Experiment in Compiler Construction Semantic Analysis (2)

Nguyen Huu Duc

Department of information systems
Faculty of information technology
Hanoi university of technology

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 analysed, 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, 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`
 - Return type, in function `compileType`
- 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`).

Project organization

#	Filename	Task
1	Makefile	Project
2	scanner.c, scanner.h	Token reader
3	reader.h, reader.c	Read character from source file
4	charcode.h, charcode.c	Classify character
5	token.h, token.c	Recognize and classify token, keywords
6	error.h, error.c	Manage error types and messages
7	parser.c, parser.h	Parse programming structure
8	debug.c, debug.h	Debugging
9	symtab.c symtab.h	Symbol table construction
10	main.c	Main program

Assignment 2

- Observe the structure of parser (modified)
- Complete *TODO* function
- Test on provided examples

Example

- Insert information of a constant
- Assignment 1

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


void compileBlock(void)

```
{ Object* constObj;
  ConstantValue* constValue;
  if (lookAhead->tokenType == KW_CONST) {
    eat(KW_CONST);
    do {
      eat(TK_IDENT);
      constObj = createConstantObject(currentToken->string);
      eat(SB_EQ);
      constValue = compileConstant();
      constObj->constAttrs->value = constValue;
      declareObject(constObj);
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);
    compileBlock2();
  }
  else compileBlock2();
}
```

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

Experiment in Compiler Construction Semantic Analysis (3)

Nguyen Huu Duc

Department of information systems
Faculty of information technology
Hanoi university of technology

Overview

- Checking duplicate object declaration
- Checking reference to object

Checking fresh identifier

- A fresh identifier is an identifier that is new (has not been used) in current scope
- Checking fresh identifier is task of function

```
void checkFreshIdent(char *name);
```

Checking fresh identifier

- Checking fresh identifier is performed in
 - Constant declaration
 - User-defined type declaration
 - Variable declaration
 - Parameter declaration
 - Function declaration
 - Procedure declaration

Checking declared constant

- Performed when there is a reference to a constant, e.g:
 - When analysing an unsigned constant
 - When analysing an constant
- If a constant is not declared in current block, search in outer blocks.
- The value of declared constant will be the value of the constant that we are dealing with
 - Share the value
 - Do not share the value →
`duplicateConstantValue`

Checking declared type

- Performed when there is a reference to a type, e.g: when analysing a type in function `compileType`
- If a type is not declared in current block, search in outer blocks
- The actual type of referred type name will be used to create the type we are dealing with
 - Share type
 - Do not share type → `duplicateType`

Checking declared variable

- Performed when there is a reference to a variable, e.g:
 - In assign statement
 - In for statement
 - When analysing factor
- If a variable is not declared in current block, search in outer blocks.

Checking declared variable

- An identifier that appears in the left-hand side of an assign statement or in a factor possibly is:
 - Current function
 - A declared variable
 - If the variable's type is array type, the array index must follow the variable's name.
- Variable is different from parameters and current function.

Checking declared function

- Performed when a function is referred, e.g
 - As left-hand side of assign statement (current function)
 - In a factor (a list of parameters will follow function's name)
- If a function is not declared in current block, search in outer blocks.
- Global functions: READC, READI

Checking a declared procedure

- Performed when a procedure is referred, e.g:
 - In CALL statement
- If a procedure is not declared in current block, search in outer blocks.
- Global procedures: WRITEI, WRITEC, WRITELN

List of error codes

- ERR_UNDECLARED_IDENT
- ERR_UNDECLARED_CONSTANT
- ERR_UNDECLARED_TYPE
- ERR_UNDECLARED_VARIABLE
- ERR_UNDECLARED_FUNCTION
- ERR_UNDECLARED_PROCEDURE
- ERR_DUPLICATE_IDENT

Project organization

#	Filename	Task
1	Makefile	Project
2	scanner.c, scanner.h	Token reader
3	reader.h, reader.c	Read character from source file
4	charcode.h, charcode.c	Classify character
5	token.h, token.c	Recognize and classify token, keywords
6	error.h, error.c	Manage error types and messages
7	parser.c, parser.h	Parse programming structure
8	debug.c, debug.h	Debugging
9	symtab.c symtab.h	Symbol table construction
10	semantics.c. semantics.h	Analyse the program's semantic
11	main.c	Main program

Assignment 3

- Implement the following function in *semantics.c*
 - `checkFreshIdent`
 - `checkDeclaredIdent`
 - `checkDeclaredConstant`
 - `checkDeclaredType`
 - `checkDeclaredVariable`
 - `checkDeclaredProcedure`
 - `checkDeclaredLValueIdent`
- Test on provided examples

Experiment in Compiler Construction Semantic Analysis (4)

Nguyen Huu Duc

Department of information systems
Faculty of information technology
Hanoi university of technology

Overview

- Type checking
- Checking the consistency between the declaration and usage of arrays.
- Checking the consistency between the declaration and usage of functions.
- Checking the consistency between the declaration and calling of procedures.
- Checking the consistency in reference usage

Type checking

- Type comparison
 - `checkIntType`
 - `checkCharType`
 - `checkArrayType`
 - `checkTypeEquality`

Type checking

- Constant:
 - [+/-] <constant>
 - The type of <constant> is integer

Type checking

- Assign statement
 - $\langle \text{LValue} \rangle := \langle \text{Expr} \rangle;$
 - Basic types of $\langle \text{Lvalue} \rangle$ and $\langle \text{Expr} \rangle$ must be the same

Type checking

- For statement:
 - For $\langle \text{var} \rangle := \langle \text{exp1} \rangle$ To $\langle \text{exp2} \rangle$ do $\langle \text{stmt} \rangle$
 - Basic types of $\langle \text{var} \rangle$, $\langle \text{exp1} \rangle$, and $\langle \text{exp2} \rangle$ must be the same

Type checking

- Function and procedure:
 - Types of declared parameter and actual parameter must be the same
 - The corresponding actual parameter of a variable declared parameter must be a LValue.

Type checking

- Condition:
- $\langle \text{exp1} \rangle \langle \text{op} \rangle \langle \text{exp2} \rangle$
 - The basic types of $\langle \text{exp1} \rangle$ and $\langle \text{exp2} \rangle$ must be the same

Type checking

- Expression:

[+|-] <exp> \rightarrow <exp> : integer

[*|/] <term> \rightarrow <term> : integer

Type checking

- Index:
- $(. \text{<exp> } .) \rightarrow \text{<exp> : integer}$
- The number of dimension of the array must be considered

Project organization

#	Filename	Task
1	Makefile	Project
2	scanner.c, scanner.h	Token reader
3	reader.h, reader.c	Read character from source file
4	charcode.h, charcode.c	Classify character
5	token.h, token.c	Recognize and classify token, keywords
6	error.h, error.c	Manage error types and messages
7	parser.c, parser.h	Parse programming structure
8	debug.c, debug.h	Debugging
9	symtab.c symtab.h	Symbol table construction
10	semantics.c. semantics.h	Analyse the program's semantic
11	main.c	Main program

Assignment 4

- Implement the following function in *semantic.c*
 - `void checkIntType (Type* type);`
 - `void checkCharType (Type* type);`
 - `void checkArrayType (Type* type);`
 - `void checkBasicType (Type* type);`
 - `void checkTypeEquality (Type*
type1, Type* type2);`

Structure for types

```
struct Type_ {  
    enum TypeClass typeClass;  
    int arraySize;  
    struct Type_ *elementType;  
};
```

```
enum TypeClass {  
    TP_INT,  
    TP_CHAR,  
    TP_ARRAY  
};
```

Assignment 4

- Update *parser.c* with the implementation of described type checking rules
- Test on provided examples