**Chap 1: *Experiment in Compiler - ConstructionScanner design***

***Define:***The compiler’s component/module that perform the job of lexical analysis (scanning) is called scanner.

**Tasks:**

* Skip meaningless characters: blank, tab, new line character, comment.
* Recognize illegal character and return error message
* Recognize different types of token
* Recognize tokens of different types
* Pass recognized tokens to the *parser* (the module that perform the job of syntatic analysis)

**Include:**

|  |  |  |
| --- | --- | --- |
| **#** | **Filename** | **Task** |
| 1 | Makefile | Project |
| 2 | scanner.c | Main |
| 3 | reader.h, reader.c | Read the source code |
| 4 | charcode.h, charcode.c | Classify character |
| 5 | token.h, token.c | Classify and recognize token, keywords |
| 6 | error.h, error.c | Manage error types and messages |

**Importan functions:**

* + **void skipBlank();**
  + **void skipComment();**
  + **Token\* readIdentKeyword(void);**
  + **Token\* readNumber(void);**
  + **Token\* readConstChar(void);**
  + **Token\* getToken(void);**

**Chap 2: *Experiment in Compiler Construction Parcer design***

***Tasks***

* Check the syntactic structure of a given program
  + Syntactic structure is given by Grammar
* Invoke semantic analysis and code generation
  + In an one-pass compiler, this module is very important since this forms the skeleton of the compiler

|  |  |  |
| --- | --- | --- |
| **#** | **Filename** | **Task** |
| 1 | Makefile | Project |
| 2 | scanner.c | Main |
| 3 | reader.h, reader.c | Read the source code |
| 4 | charcode.h, charcode.c | Classify character |
| 5 | token.h, token.c | Classify and recognize token, keywords |
| 6 | error.h, error.c | Manage error types and messages |
| 7 | parcer.h, parcer.c | Check the structure of the programe |

**Important functions:**

|  |  |  |  |
| --- | --- | --- | --- |
| compileProgram | compileBlock | compileBlock2 | compileBlock3 |
| compileBlock4 | compileBlock5 | compileConstDecls | compileConstDecl |
| compileTypeDecl | compileTypeDecls | compileVarDecls | compileVarDecl |
| compileSubDecls | compileFunDecl | compileProcDecl | compileParms |
| compileParams2 | compileType | CompileBasicType | compileUnsignedConstant |
| CompileConstant |  |  |  |

**CHAPTER 4. DESIGN SEMANTIC ANALYZER FOR KPL**

* 1. **Tasks of semantic analyzer**

Import tasks of a sematic analyser:

- Produce symbol table for future references (eg. scope and type checking).

-Scope checking

-Type checking.

* 1. **Design symbol table**
     1. **Why we need symbol table**

We need a symbol table to store information needed about every identifiers in the program.

Each [identifier](http://en.wikipedia.org/wiki/Identifier) in a program's [source code](http://en.wikipedia.org/wiki/Source_code) is associated with information relating to its declaration or appearance in the source, such as its [type](http://en.wikipedia.org/wiki/Data_type), [scope](http://en.wikipedia.org/wiki/Scope_(programming)) level and its [location](http://en.wikipedia.org/wiki/Memory_address).

* + 1. **Design symbol table**

**4.2.2.1.Data structure in symbol table**

struct SymTab\_ { }

The data structure to represent symbol table itself, including:

* Program: the program object.
* currentScope: current scope of symbol table.
* globalObjectList: store global objects such as functions: CALLI, WRITEI, etc.

struct Object\_ {

char name[MAX\_IDENT\_LEN];

enum ObjectKind kind;

union {

};

To store information about each object in program, such as main program itself, a procedure or function, a variable, a constant, etc.

enum ObjectKind { };

Kinds of object (a variable object, a constant object, etc)

enum ParamKind { };

Kinds of parameter: value or reference type.

struct Scope\_ {};

To store information about a scope, including:

* objList: objects in the scope.
* Owner: function or procedure has that scope.
* Outer: the outer scope of it.

struct ObjectNode\_ {};

A linked list to represent list of objects.

typedef struct ConstantAttributes\_ ConstantAttributes;

typedef struct TypeAttributes\_ TypeAttributes;

typedef struct VariableAttributes\_ VariableAttributes;

typedef struct FunctionAttributes\_ FunctionAttributes;

typedef struct ProcedureAttributes\_ ProcedureAttributes;

typedef struct ProgramAttributes\_ ProgramAttributes;

typedef struct ParameterAttributes\_ ParameterAttributes;

To store typical attributes of each type: e.g a constant type must have a constant value, a function type must have a parameter list, a return type, and own a scope.

**4.2.2.2.Functions in symbol table.**

Object\* createProgramObject(char \*programName): create a program object.

Object\* createConstantObject(char \*name): create a constant object.

Object\* createTypeObject(char \*name): create a type object.

Object\* createVariableObject(char \*name): create a variable object.

Object\* createFunctionObject(char \*name): create a function object.

Object\* createProcedureObject(char \*name): create a procedure object.

Object\* createParameterObject(char \*name, enum ParamKind kind, Object\* owner): create a parameter object.

Type\* makeIntType(void): create an integer type.

Type\* makeCharType(void): create a character type.

Type\* makeArrayType(int arraySize, Type\* elementType): create an array type.

Type\* duplicateType(Type\* type): copy type.

int compareType(Type\* type1, Type\* type2): compare type

ConstantValue\* makeIntConstant(int i): create an integer constant.

ConstantValue\* makeCharConstant(char ch): create a character constant.

ConstantValue\* duplicateConstantValue(ConstantValue\* v): copy a constant.

Scope\* createScope(Object\* owner, Scope\* outer): create a scope.

Object\* findObject(ObjectNode \*objList, char \*name): find object with specific name in an object list.

* 1. **Verify scoping rules**
     1. **Checking fresh identifier**

We determine if an identifier is not declared yet, by function void checkFreshIdent(char \*name)

void checkFreshIdent(char \*name) {

if (findObject(symtab->currentScope->objList, name) != NULL)

error(ERR\_DUPLICATE\_IDENT, currentToken->lineNo, currentToken->colNo);

}

If the identifier is already declared, findObject function will return an non-null value.

* + 1. **Checking declared identifier**

Object\* checkDeclaredIdent(char \*name): check declared identifiers: (identifier is already declared or not. If declared return identifier object, else return null.)

Object\* checkDeclaredConstant(char \*name): check declared constants.

Object\* checkDeclaredType(char \*name): check declared identifiers.

Object\* checkDeclaredVariable(char \*name): check declared variables.

Object\* checkDeclaredFunction(char \*name): check declared functions.

Object\* checkDeclaredProcedure(char \*name): check declared procedure.

Object\* checkDeclaredLValueIdent(char \*name): check declared LValue.

* 1. **Type checking**
     1. **Checking the consistency between declaration and usage of identifiers.**

void checkIntType(Type\* type): check if type is integer

void checkCharType(Type\* type): check if type is character

void checkArrayType(Type\* type): check if type is array type.

void checkBasicType(Type\* type): check if type is basic type.

void checkTypeEquality(Type\* type1, Type\* type2): check for equality of types, if not, report an error message.

**4.4.2. Checking specific requirements in some statement (e.g. LValue in assign statement**