Compiler Construction Report

Instructor: Nguyen thi Thu Huong

Nguyen Ngoc Lam
ICT.02-K61

University Name: Hanoi University of Science and Technology
Date: July 22, 2019

Contents

Chapter 1

AN OVERVIEW OF COMPILER

1.1 Task of a compiler

In simple words, A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for converting a source code is to create an executable program. Another critical goal of a compiler is to report errors in source code to developers.
See more at: `http://en.wikipedia.org/wiki/Compiler`

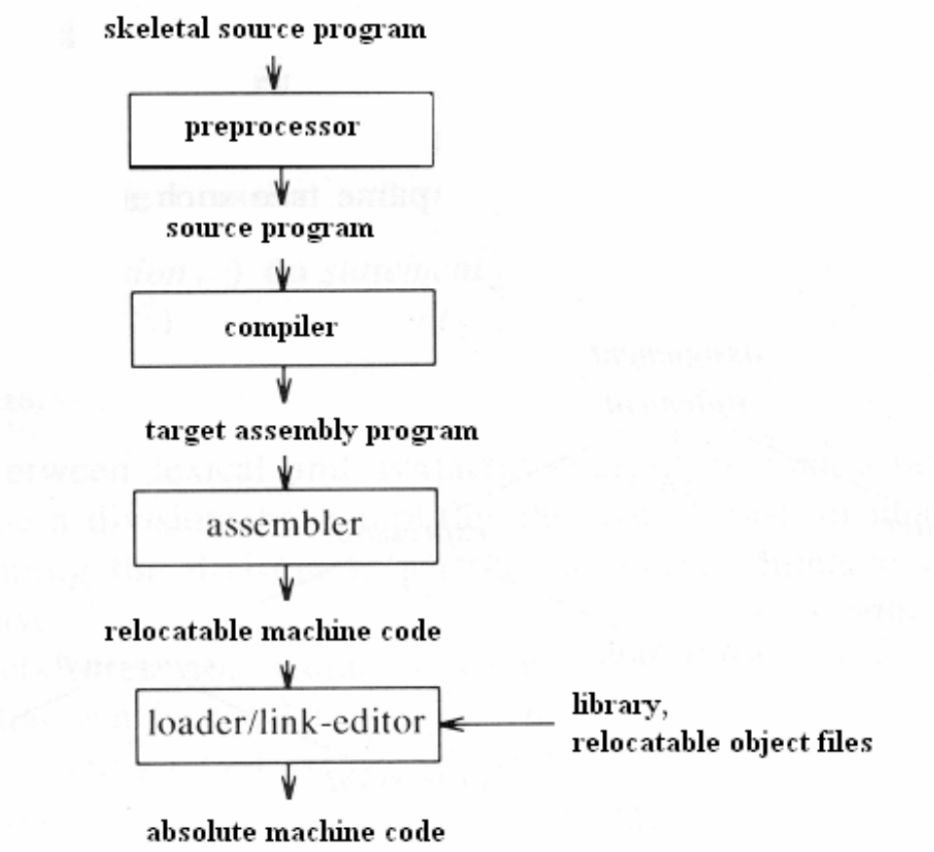


Figure 1.1: Context of a compiler in a language processor

1.2 COMPONENTS OF A COMPILER

A compiler can be divided into 4 main parts:

- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Code generator

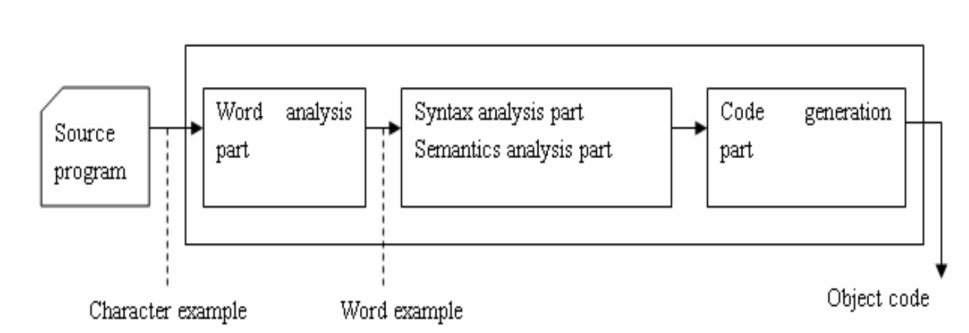


Figure 1.2: Parts of a compiler

1.3 Steps of the compiling process

A compiler is divided into several interrelated processes, in each process, source program is translated from a specific form to another form of representation. For better understanding, see figure 1.3
Consider an example in KPL:

$$Sum := initial + increment * 50 \tag{1.1}$$

1.3.1 Lexical analyzer

Lexical analysis [1] is the process of converting a sequence of characters into a sequence of tokens, i.e. meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer,

[1]Note that in the process of lexical analysis, the space, tabulator, and the comments (in KPL // or /**/) will be neglected.

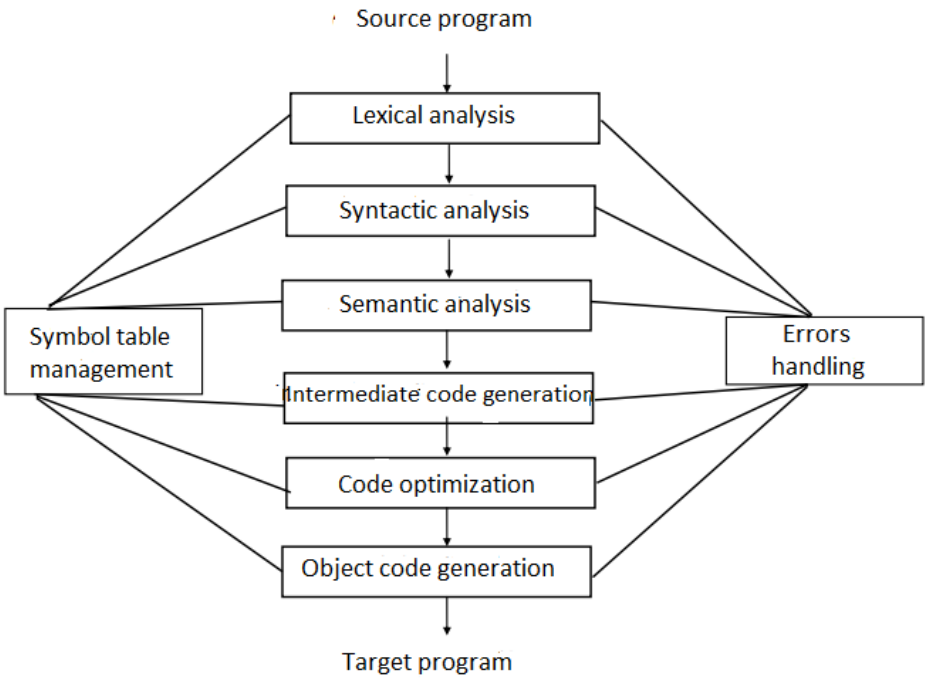


Figure 1.3: A typical decomposition

tokenizer, or scanner, though "scanner" is also used for the first stage of a lexer.
A token is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called tokenization. The characters that form a token are called a lexeme. See more at `http://en.wikipedia.org/wiki/Lexical_analysis`
The process of lexical analysis will occur as follows: the scanner will read character-by-character input stream to generate tokens. So the result of the example 1.1 will be:

1. Identifier (sum)
2. Symbol that represents assignment (:=)
3. Identifier (initial)
4. Symbol that represents addition (+)
5. Identifier (increment)
6. Symbol that represents multiplication (*)
7. Number (50)

1.3.2 Syntax analysis

Syntactic analysis (also called parsing) is the process of analysing a string of tokens, conforming to the rules of a formal grammar or not. The program that performs parsing is called the syntactic analyser or simply parser. The output of parsing is the parse tree, or error. Parsing is based on grammar provided to build parse tree. The most important part of building a compiler is the task of building a grammar that generates structure of a program and cannot be ambiguous. An ambiguous grammar will produce more than one parse tree, therefore must be forbidden.
The example 1.1 will produce following parse tree:

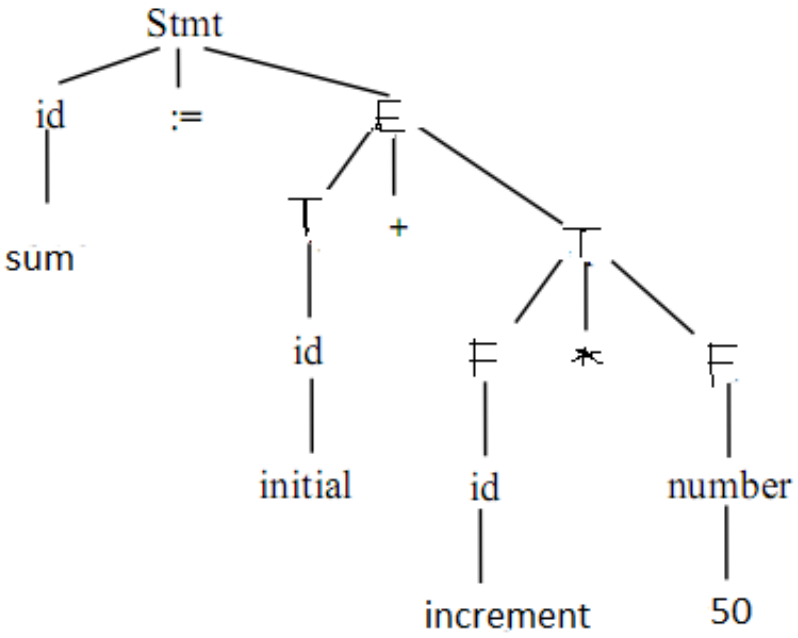


Figure 1.4: Parse tree of 1.1

1.3.3 Semantic analysis

Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase.

An important part of semantic analysis is type checking and variable scope checking. In this step, compiler will check, according to specification from source language. In additional, semantic analyser will use symbol table to store information about every identifier, which will generate information about the position of storing identifiers,type and scope of them in program or procedure that include them, or if an identifier is name of function or procedure, it will store information about number and type of parameters, return type, etc.

1.3.4 Intermediate code generation

After the phase of semantic analysis, some compiler will generate an intermediate representation of source program, known as intermedia code. We can consider this representation as a program for an abstract virtual machine. They have three important properties: easy to generate, easy to translate into object code and machine-independent. Usually, compiler use three-address codes. Three-address codes are codes that accept at most three parameters, one operator (except assignment). So before generation of these codes, compiler need rules of operator precedence, e.g * before +. Example 1.1 gives the following intermediate code:

$$T1 := 50;$$

$$T2 := ID3 * T1;$$

$$T3 := ID2 + T2;$$

$$ID1 := T3;$$

1.3.5 Code optimization

In this phase, code optimizator will try to optimize the intermediate code into equivalent one with faster execution. For example, the example 1.1 can be optimized as:

$$T1 := ID3 * 50;$$

$$ID1 := ID2 + T1;$$

There is a significant difference between the amount of optimization codes done by different compiler. In some compiler called optimize-focused compiler, a conspicuous proportion of time devoted to this phase. However, there are also some optimization method that can decrese lots of execution time of source program without wasting too much time compiling.

1.3.6 Object code generation

This is the final phase of compiler. Input of a object code generator is the intermediate code and output is the target program. There are a number of factors that affect the design of an object code generator such as: memory management, resource allocation and the sequence of code execution.

1.4 SUMMARY

In order for a computer to understand and execute a program written in a high-level programming language, we need a compiler to translate source program to target program in object codes. This chapter has presented an overview of a compiler in general, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and object code generation. Output of the preceding phases are always input of the following phases, i.e, output of a scanner (tokens) will be input of the parser, output of the parser (parse tree) will be input of the semantic analyzer, etc.

Chapter 2

LEXICAL ANALYSER FOR KPL

In a compiler, the program that perform lexical analysis is called the scanner.

2.1 TASK OF A SCANNER

- Neglect meaningless character: space, tabulalor, EOF, CR, LF, comments.
- Detect invalid symbols: @, ! (stand-alone), etc
- Detect and produce tokens: identifiers, keywords, numbers, literals, special characters, etc.

2.2 TOKENS IN KPL

- Identifier: variable name, constant names, type names, function names, procedure names:
  - Start with letter or underscore: a-z, A-Z, '_'
  - Others are letter, underscore or numbers
- Keywords: PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL, IF, ELSE, WHILE, DO, FOR, TO
- Operators: := (assign), + (addition), - (subtraction), * (multiplication), / (division), = (comparison of equality), != (comparison of difference), > (comparison of greaterness), < (comparison of lessness), >= (comparison of greaterness or equality), <= (comparison of lessness or equality)
- Special characters ; (semicolon), . (period), : (colon), , (comma), ((left parenthesis),) (right parenthesis), ´ (singlequote), (. and .) to specify indexes in arrays and (*, *) to indicate comments

- Others: integer number, string literals,...

2.3 DATA STRUCTURE IN KPL

1. **Data structure to store valid characters in KPL :**
   space, *letters*, *numbers*, +, −, ∗, /, <, >, !, =, , , ., :, ; , , (., .)
   others are invalid characters (CHAR_ UNKNOWN)

   typedef enum {
   CHAR_ SPACE, CHAR_ LETTER, CHAR_ DIGIT, CHAR_ PLUS,
   CHAR_ MINUS, CHAR_ TIMES, CHAR_ SLASH, CHAR_ LT,
   CHAR_ GT, CHAR_ EXCLAIMATION, CHAR_ EQ, CHAR_ COMMA,
   CHAR_ PERIOD, CHAR_ COLON, CHAR_ SEMICOLON,
   CHAR_ SINGLEQUOTE, CHAR_ LPAR, CHAR_ RPAR,
   CHAR_ UNKNOWN
   } CharCode;

2. **Stores token types in KPL:**

   typedef enum {
   TK_ NONE, TK_ IDENT, TK_ NUMBER, TK_ CHAR, TK_ EOF,
   KW_ PROGRAM, KW_ CONST, KW_ TYPE, KW_ VAR,
   KW_ INTEGER, KW_ CHAR, KW_ ARRAY, KW_ OF,
   KW_ FUNCTION, KW_ PROCEDURE,
   KW_ BEGIN, KW_ END, KW_ CALL,
   KW_ IF, KW_ THEN, KW_ ELSE,
   KW_ WHILE, KW_ DO, KW_ FOR, KW_ TO,

   SB_ SEMICOLON, SB_ COLON, SB_ PERIOD, SB_ COMMA,

SB_ ASSIGN, SB_ EQ, SB_ NEQ, SB_ LT, SB_ LE, SB_ GT,
SB_ GE,
SB_ PLUS, SB_ MINUS, SB_ TIMES, SB_ SLASH,
SB_ LPAR, SB_ RPAR, SB_ LSEL, SB_ RSEL
} TokenType;

3. **Store information about each tokens:**

   - string : content of token
   - lineNo , colNo : position of token,
   - tokenType : type of token
   - value : value of token if a number.

```
typedef struct {
char string[MAX_ IDENT_ LEN + 1];
int lineNo, colNo; // line and column of tokens
TokenType tokenType;
int value;
} Token;
```

2.4 FUNCTIONS IN KPL

2.4.1 Details about functions

- **void** skipBlank() : skip spaces.
- **void** skipComment() : skip comments.
- **Token*** readIdentKeyword() : read identifiers or keywords, return a pointer of Token type.
- **Token*** readNumber() : read a integer number, return a pointer of Token type.
- **Token*** readConstChar() : read a constant character, return a pointer of Token type.

- **TokenType** checkKeyword(**char** ***string) : check if the string is a keyword, return TOKEN_ NONE if keyword.
- **Token*** makeToken(**TokenType** tokenType, **int** lineNo, **int** colNo) : create a pointer to a token with predefined type and position.
- **Token*** getToken() : read and return a token (can be invalid token: TOKEN_ NONE).
- **Token*** getValidToken() : read and return a valid token.

2.4.2 Details about execution of a scanner

1. Scanner is a finite automation. Everytime it return a token, the state will be 0. When detects invalid characters, the state will be -1
2. During the reading of input stream, getToken() will be looped until meet (EOF)
3. If detect
   - spaces (CHAR_ SPACE) -> skipBlank(), state will be 0, -> getToken(), ...
   - other tokens, that token will be passed to parser for the next phase of compiling process.

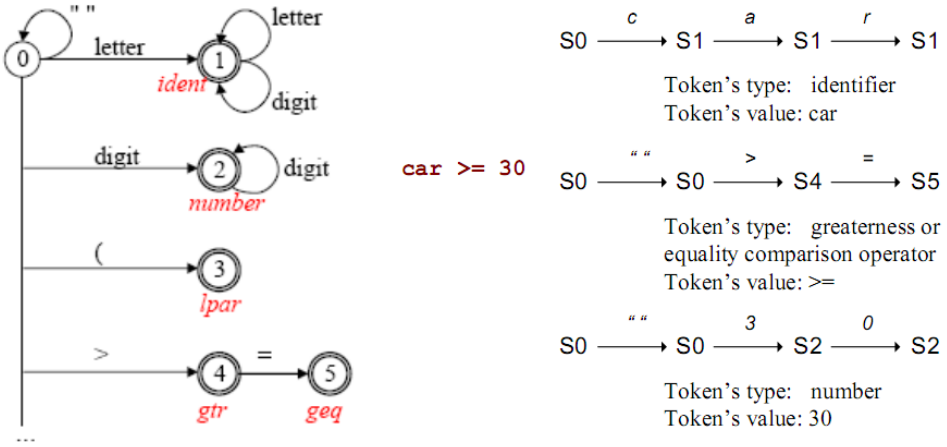


Figure 2.1: Execution of a scanner

Chapter 3

SYNTACTIC ANALYSER FOR KPL

3.1 Task of a syntatic analyser (parser)

- Check the syntax of the program for errors
- Produce parse tree for semantic analyser otherwise

3.2 Syntax diagram and BNF grammar

3.2.1 Syntax diagram

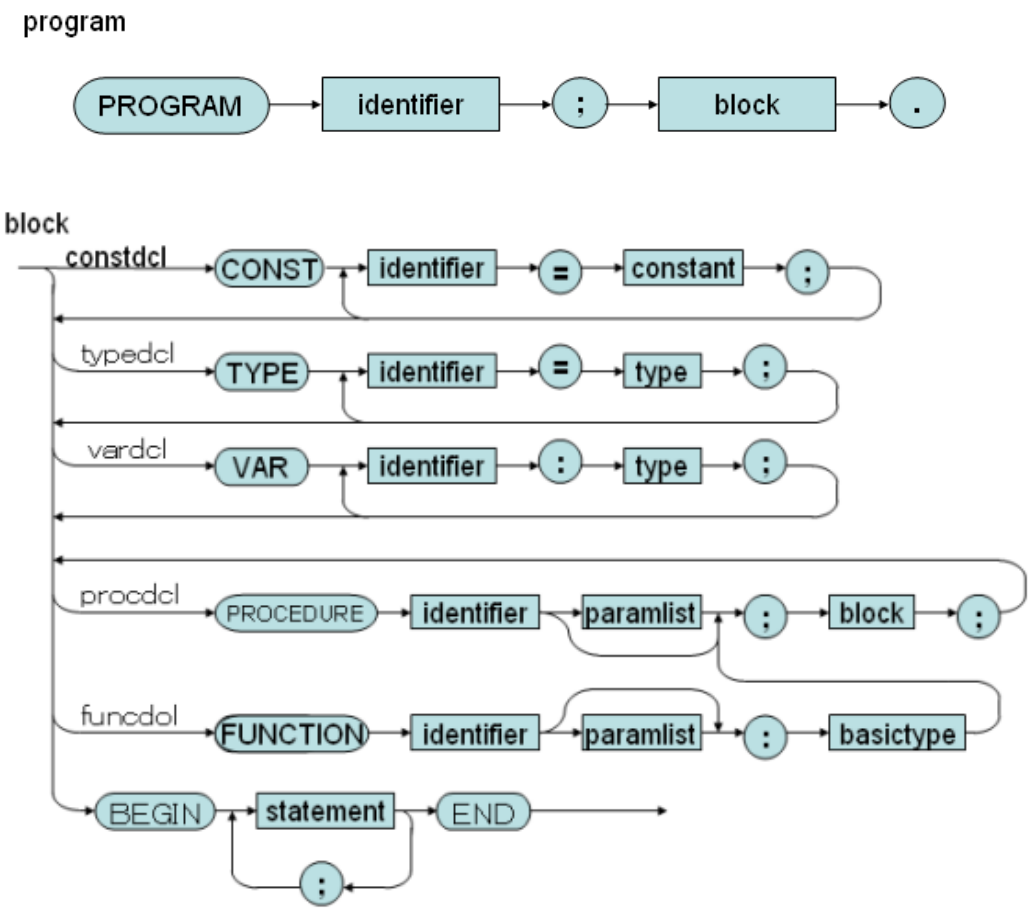
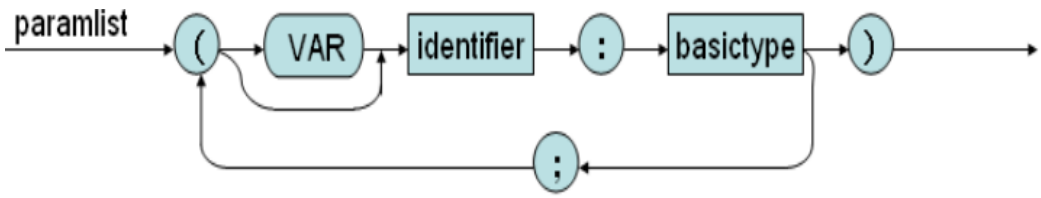
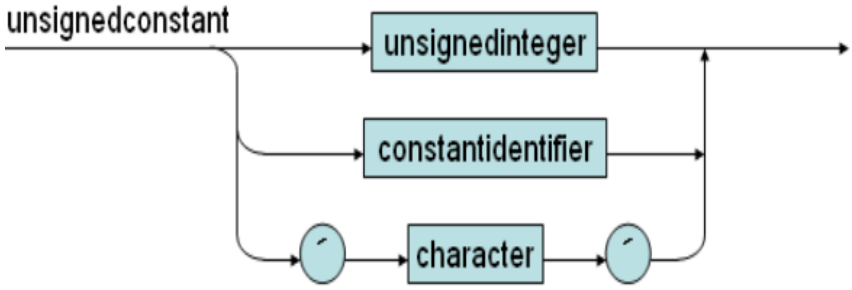


Figure 3.1: Syntax diagram 1.





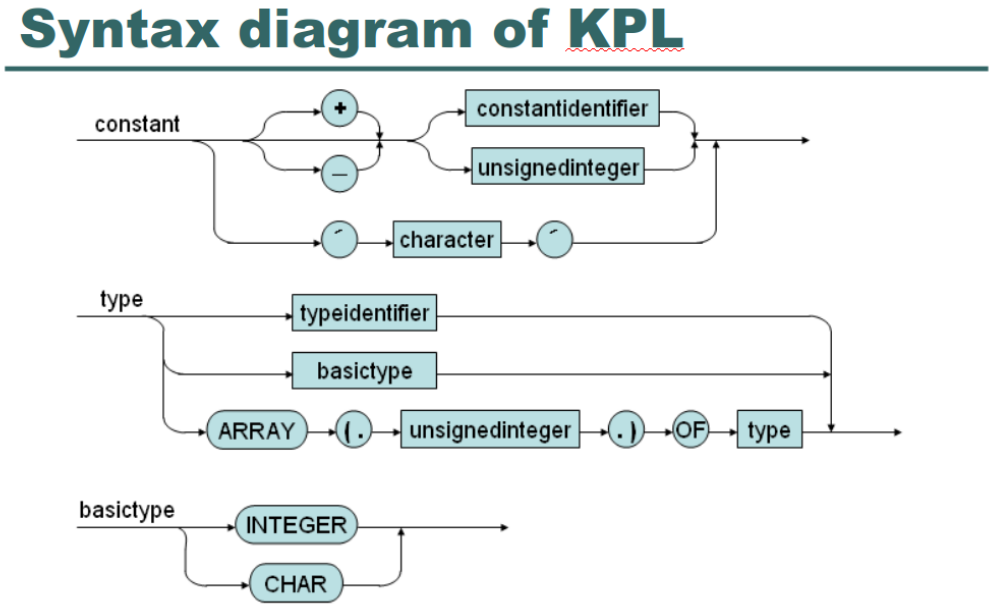


Figure 3.2: Syntax diagram 2.

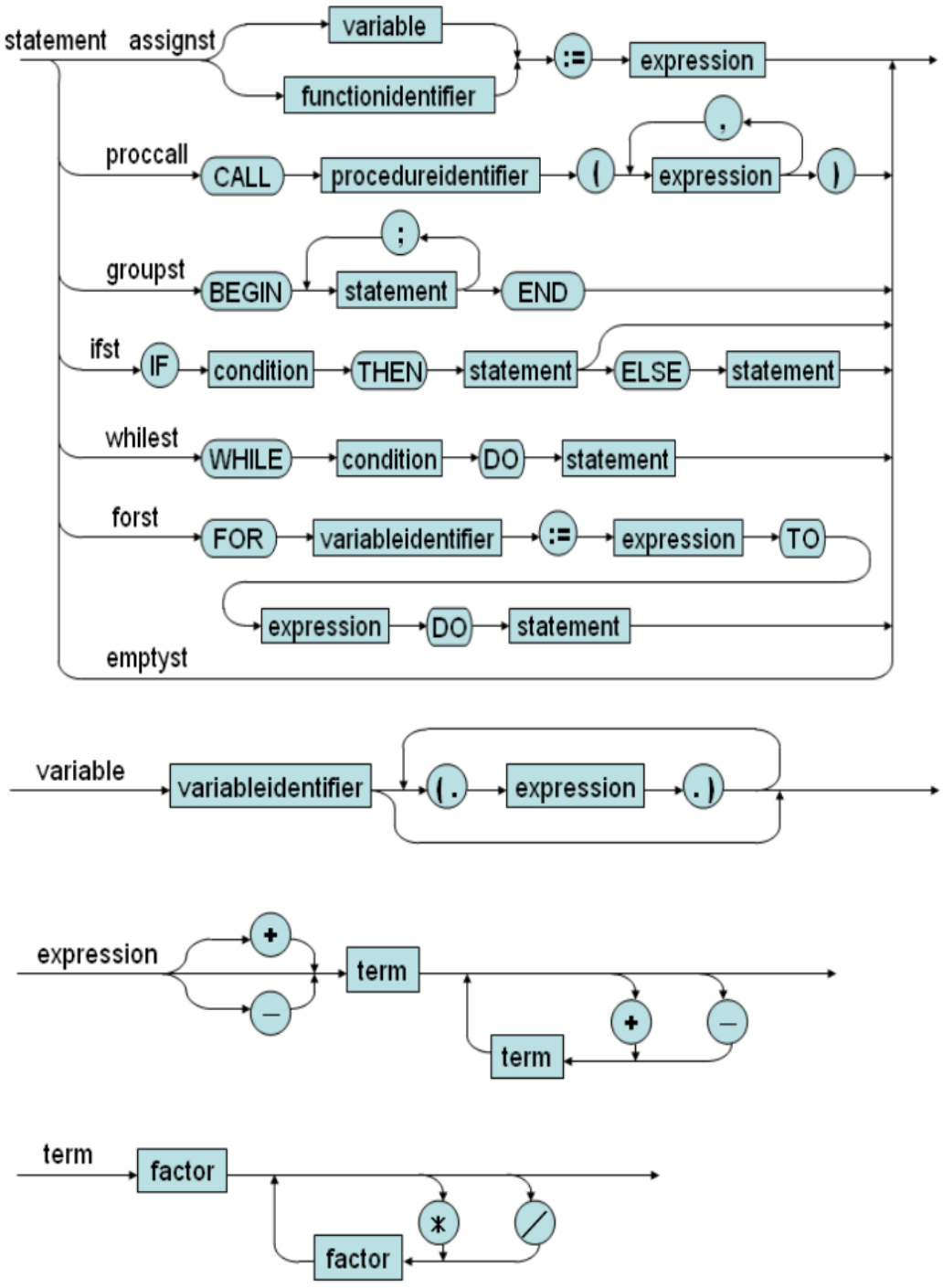


Figure 3.3: Syntax diagram 3.

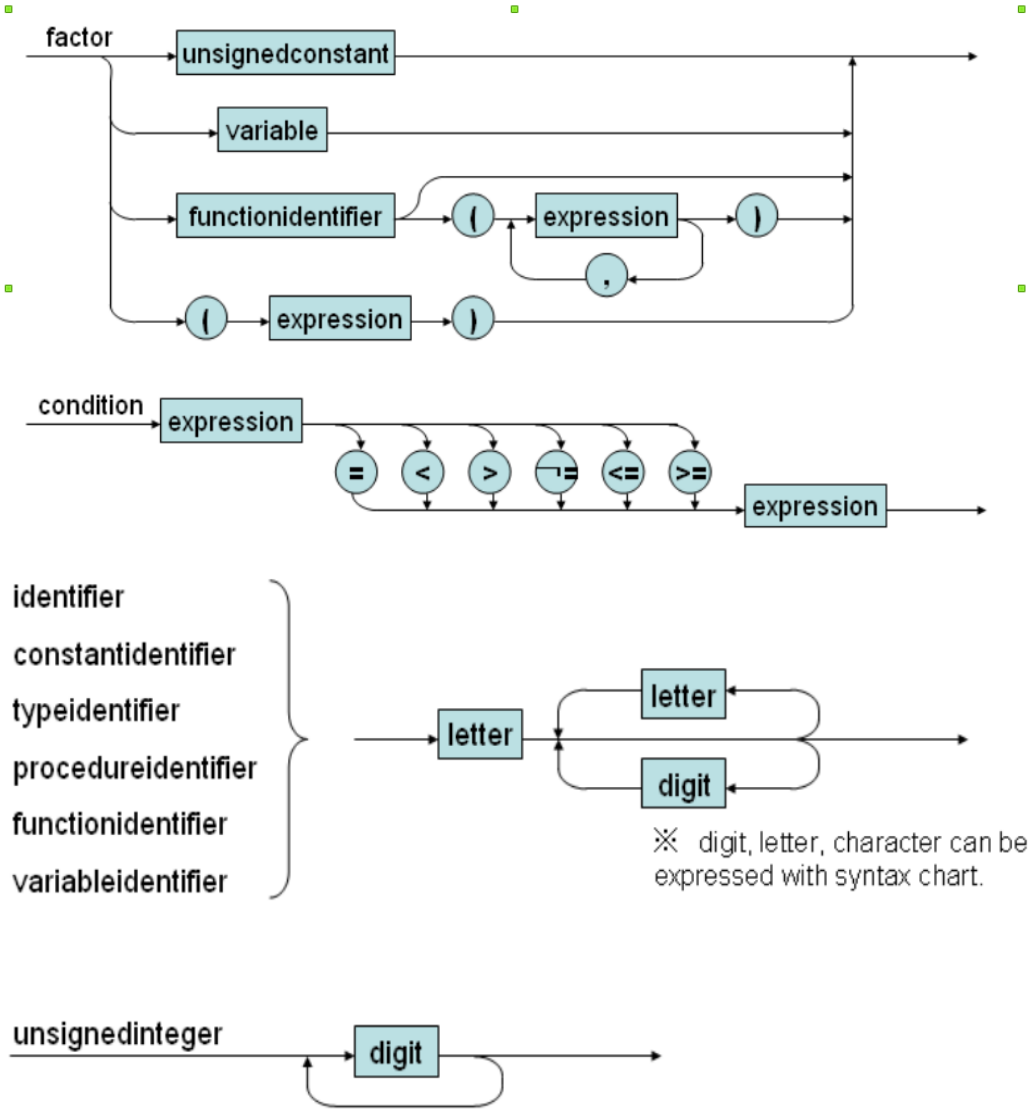


Figure 3.4: Syntax diagram 4.

3.2.2 BNF grammar

1. Prog ::= KW_PROGRAM Ident SB_SEMICOLON Block SB_PERIOD
2. Block ::= KW_CONST ConstDecl ConstDecls Block2
3. Block ::= Block2
4. Block2 ::= KW_TYPE TypeDecl TypeDecls Block3
5. Block2 ::= Block3
6. Block3 ::= KW_VAR VarDecl VarDecls Block4
7. Block3 ::= Block4
8. Block4 ::= SubDecls Block5
9. Block5 ::= KW_BEGIN Statements KW_END
10. ConstDecls::= ConstDecl ConstDecls
11. ConstDecls::= ϵ
12. ConstDecl ::= Ident SB_EQUAL Constant SB_SEMICOLON
13. TypeDecls ::= TypeDecl TypeDecls
14. TypeDecls ::= ϵ
15. TypeDecl ::= Ident SB_EQUAL Type SB_SEMICOLON
16. VarDecls ::= VarDecl VarDecls
17. VarDecls ::= ϵ
18. VarDecl ::= Ident SB_COLON Type SB_SEMICOLON
19. SubDecls ::= FunDecl SubDecls
20. SubDecls ::= ProcDecl SubDecls

21. SubDecls ::= ϵ
22. FunDecl ::= KW_FUNCTION Ident Params SB_COLON BasicType SB_SEMICOLON Block SB_SEMICOLON
23. ProcDecl ::= KW_PROCEDURE Ident Params SB_SEMICOLON Block SB_SEMICOLON
24. Params ::= SB_LPAR Param Params2 SB_RPAR
25. Params ::= ϵ
26. Params2 ::= SB_SEMICOLON Param Params2
27. Params2 ::= ϵ
28. Param ::= Ident SB_COLON BasicType
29. Param ::= KW_VAR Ident SB_COLON BasicType
30. Type ::= KW_INTEGER
31. Type ::= KW_CHAR
32. Type ::= TypeIdent
33. Type ::= KW_ARRAY SB_LSEL Number SB_RSEL KW_OF Type
34. BasicType ::= KW_INTEGER
35. BasicType ::= KW_CHAR
36. UnsignedConstant ::= Number
37. UnsignedConstant ::= ConstIdent
38. UnsignedConstant ::= ConstChar
39. Constant ::= SB_PLUS Constant2
40. Constant ::= SB_MINUS Constant2

41. Constant ::= Constant2
42. Constant ::= ConstChar
43. Constant2::= ConstIdent
44. Constant2::= Number
45. Statements ::= Statement Statements2
46. Statements2 ::= KW_SEMICOLON Statement Statement2
47. Statements2 ::= ϵ
48. Statement ::= AssignSt
49. Statement ::= CallSt
50. Statement ::= GroupSt
51. Statement ::= IfSt
52. Statement ::= WhileSt
53. Statement ::= ForSt
54. Statement ::= ϵ
55. AssignSt ::= Variable SB_ASSIGN Expession
56. AssignSt ::= FunctionIdent SB_ASSIGN Expression
57. CallSt ::= KW_CALL ProcedureIdent Arguments
58. GroupSt ::= KW_BEGIN Statements KW_END
59. IfSt ::= KW_IF Condition KW_THEN Statement ElseSt
60. ElseSt ::= KW_ELSE statement
61. ElseSt ::= ϵ
62. WhileSt ::= KW_WHILE Condition KW_DO Statement

63. ForSt ::= KW_FOR VariableIdent SB_ASSIGN Expression KW_TO Expression KW_DO Statement
64. Arguments ::= SB_LPAR Expression Arguments2 SB_RLAR
65. Arguments ::= ϵ
66. Arguments2::= SB_COMMA Expression Arguments2
67. Arguments2::= ϵ
68. Condition ::= Expression Condition2
69. Condition2::= SB_EQ Expression
70. Condition2::= SB_NEQ Expression
71. Condition2::= SB_LE Expression
72. Condition2::= SB_LT Expression
73. Condition2::= SB_GE Expression
74. Condition2::= SB_GT Expression
75. Expression ::= SB_PLUS Expression2
76. Expression ::= SB_MINUS Expression2
77. Expression ::= Expression2
78. Expression2 ::= Term Expression3
79. Expression3 ::= SB_PLUS Term Expression3
80. Expression3 ::= SB_MINUS Term Expression3
81. Expression3 ::= ϵ
82. Term ::= Factor Term2
83. Term2 ::= SB_TIMES Factor Term2

84. Term2 ::= SB_SLASH Factor Term2
85. Term2 ::= ϵ
86. Factor ::= UnsignedConstant
87. Factor ::= Variable
88. Factor ::= FunctionApptication
89. Factor ::= SB_LPAR Expression SB_RPAR
90. Variable ::= VariableIdent Indexes
91. FunctionApplication ::= FunctionIdent Arguments
92. Indexes ::= SB_LSEL Expression SB_RSEL Indexes
93. Indexes ::= ϵ

3.3 Recursive descent parsing

- Properties:
  - LL(k) is the language that needs looking ahead k character to produce a valid production
  - Used to parse LL(1) language
  - Can be extended for LL(k), but very complex
  - Used for other grammar can lead to infinite iteration
- Recursive descent parsing:
  - A top-down parsing method.
  - The term descent refers to the direction in which the parse tree is traversed (or built).

- Use a set of mutually recursive procedures (one procedure for each nonterminal symbol). Start the parsing process by calling the procedure that corresponds to the start symbol. Each production becomes one clause in procedure
- Consider a special type of recursive-descent parsing called predictive parsing. Use a lookahead symbol to decide which production.

3.4 Data structure in parser for KPL

Like in scanner.

3.5 Parse terminal symbols

void eat(**TokenType** tokenType);
Function will compare the passed tokenType to token type read in scanner (currentToken). If equals, print out the token, otherwise, report error: missing token at that position.

3.6 Parsing non-terminal symbols

- **void** compileProgram(): parse main program.
- **void** compileBlock(**void**): parse constant declarations then call compileBlock2.
- **void** compileBlock2(**void**): parse type declarations then call compileBlock3.
- **void** compileBlock3(**void**): parse variable declarations then call compileBlock4.

- **void** compileBlock4(**void**):parse subroutines declarations then call compileBlock5.
- **void** compileBlock5(**void**): parse statements in main function.
- **void** compileConstDecls(**void**): parse constant declarations.
- **void** compileConstDecl(**void**): parse a single constant declaration.
- **void** compileTypeDecls(**void**): parse type declarations.
- **void** compileTypeDecl(**void**): parse a single type declaration.
- **void** compileVarDecls(**void**): parse variable declarations.
- **void** compileVarDecl(**void**): parse a variable declaration.
- **void** compileSubDecls(**void**): parse subroutines declarations.
- **void** compileFuncDecl(**void**): parse function declarations.
- **void** compileProcDecl(**void**): parse procedures declarations.
- **void** compileUnsignedConstant(**void**): parse unsigned constants.
- **void** compileConstant(**void**): parse signed constants.
- **void** compileType(**void**): parse a type.
- **void** compileBasicType(**void**): parse a basic type.
- **void** compileParams(**void**): parse list of parameters.
- **void** compileParam(**void**): parse a single parameter.
- **void** compileStatements(**void**): parse all statements.

- **void** compileStatement(**void**): parse a single statement.
- **void** compileAssignSt(**void**): parse an assignment statement.
- **void** compileCallSt(**void**): parse a call statement.
- **void** compileIfSt(**void**): parse an IF statement.
- **void** compileElseSt(**void**): parse an ELSE statement.
- **void** compileWhileSt(**void**): parse a WHILE statement.
- **void** compileForSt(**void**): parse a FOR statement.
- **void** compileArguments(**void**): parse list of arguments passed to a function or procedure.
- **void** compileArguments2(**void**): parse list of arguments passed to a function or procedure.
- **void** compileCondition(**void**): parse conditional expression.
- **void** compileExpression(**void**): parse (+,-) of an expression then call compileExpression2
- **void** compileExpression2(**void**): parse (+, -) operators between terms then call compileExpression3
- **void** compileExpression3(**void**): recursive of (+, -) operators between terms.
- **void** compileTerm(**void**): compile a term, which can be composed of (*, /) of compileFactor
- **void** compileTerm2(**void**) recursive procedure of (*, /) between factors.

- **void** compileFactor(**void**): a factor can be a number, character, identifier .
- **void** compileIndexes(**void**): parse indexes of an array.

Chapter 4

SEMANTIC ANALYSER FOR KPL

4.1 Tasks of semantic analyzer

Important tasks of a semantic analyzer:

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

4.2 Symbol table designing

4.2.1 Reason

We need a symbol table to store information needed about every identifiers in the program. Each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and its location.

4.2.2 Design symbol table

Data structure in symbol table

1. struct **SymTab_** {
   Object* program;
   Scope* currentScope;
   ObjectNode *globalObjectList;
   };
   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

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

3. **Kinds of object (a variable object, a constant object, etc)**
   enum **ObjectKind** {
   OBJ_CONSTANT,
   OBJ_VARIABLE,
   OBJ_TYPE,
   OBJ_FUNCTION,
   OBJ_PROCEDURE,
   OBJ_PARAMETER,
   OBJ_PROGRAM
   };

4. **Kinds of parameter: value or reference type**
   enum **ParamKind** {

PARAM_VALUE, PARAM_REFERENCE };

5. **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 **Scope_** {
   ObjectNode *objList;
   Object *owner;
   struct **Scope_** *outer;
   };

6. **A linked list to represent list of objects**
   struct **ObjectNode_** {
   Object *object;
   struct **ObjectNode_** *next;
   };

7. **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**
   struct **ConstantValue_** {
   enum **TypeClass** type;
   union {
   int intValue;
   char charValue;
   };};

   typedef struct **ConstantValue_** ConstantValue;

   struct **Scope_**;
   struct **ObjectNode_**;

```
struct Object_;

struct ConstantAttributes_ {
ConstantValue* value;
};
struct VariableAttributes_ {
Type *type;
struct Scope_ *scope;
};

struct TypeAttributes_ {
Type *actualType;
};

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

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

struct ProgramAttributes_ {
struct Scope_ *scope;
};

struct ParameterAttributes_ {
enum ParamKind kind;
```

Type* type;
struct **Object_** *function;
};

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;

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.

4.3 Verify scoping rules

4.3.1 Checking fresh identifier

We determine if an identifier is not declared yet, by function **void** checkFreshIdent(**char** *name). If the identifier is already declared, function findObject will return an non-null value
void checkFreshIdent(**char** *name) {
if (findObject(symtab->currentScope->objList, name) != **NULL**)
error(ERR_DUPLICATE_IDENT, currentToken->lineNo, currentToken->colNo);
}

4.3.2 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

4.4 Type checking

4.4.1 Reason

- Check the consistency between declaration and usage of identifiers
- Check specific requirements in some statement (e.g. LValue in assign statement

4.4.2 Functions

- **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.