## 编译原理实验课内容设计

## 构建MIDL语言的编译器MIDLCompiler

目录

[编译原理实验课内容设计 1](#_Toc5297)

[构建MIDL语言的编译器MIDLCompiler 1](#_Toc26688)

[一、背景简介 3](#_Toc14458)

[二、任务一：MIDLCompiler词法分析和语法分析 4](#_Toc31881)

[1、MIDL的词法规则 4](#_Toc4568)

[2、 MIDL的语法规则 6](#_Toc28264)

[3、 MIDLCompiler的词法和词法分析的任务 7](#_Toc110)

[三、任务二：MIDLCompiler语义分析与代码生成 9](#_Toc7078)

[1、MIDL语言的语义分析需求 9](#_Toc15569)

[2、MIDL语言的C++ 代码生成要求 10](#_Toc3273)

[3、MIDLCompiler的语义和代码生成任务 10](#_Toc7845)

[四、Antlr4开源工具简介 11](#_Toc20277)

[五、StringTemplate模板引擎介绍 15](#_Toc2251)

[六、手工撰写词法分析程序参考案例 16](#_Toc16682)

[1、Tiny语言的词法规则定义 16](#_Toc21465)

[2、基于自动机模型的Tiny语言的词法程序 17](#_Toc26327)

[七、手工撰写语法分析程序参考案例 25](#_Toc4465)

[1、Tiny语言语法规则定义 25](#_Toc22744)

[2、语法分析程序实现步骤 26](#_Toc5939)

[2.1将文法用EBNF表示 26](#_Toc8156)

[2.2 抽象语法树设计 26](#_Toc30304)

[2.3定义数据结构 28](#_Toc11848)

[2.4实现递归下降分析程序 29](#_Toc28004)

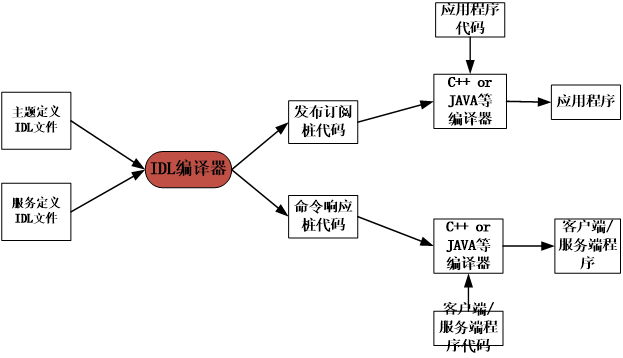
# 一、背景简介

IDL（Interface Definition Language）是一种平台无关的接口描述语言。IDL主要用于描述分布式通信的数据接口，它的语法结构简单、功能强大、跨平台，是分布式数据通信应用开发的最佳选择。IDL编译技术完成两项工作：

1）生成发布订阅需要的数据结构和桩文件；

2）生成命令响应过程中的桩代码。

IDL语言使得嵌入式中间件通信所需的数据接口和具体实现语言分离，通过IDL编译器代码生成工具，生成相应语言的接口代码，可大大提高应用程序开发效率。



本实验项目是通过裁剪IDL语言，形成MIDL（Mini Interface Definition Language），MIDL是IDL语言的一个语法子集，作为编译原理课程的实验对象。本实验项目要求研制MIDL语言的一款编译器MIDLCompiler，编译器MIDLCompiler实现从MIDL源代码转换为C++源代码的功能，被称为源到源编译器（source-to-source compiler）。在该实验项目中，要求实现MIDLCompiler源码的词法分析和语法分析，并生成抽象语法树，在此基础上进行语义分析与C++代码生成。

# 二、任务一：MIDLCompiler词法分析和语法分析

## 1、MIDL的词法规则

1. MIDL语言的关键字（20个）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| module | struct | boolean | short | long |
| unsigned | int8 | int16 | int32 | int64 |
| uint8 | uint16 | uint32 | uint64 | char |
| string | float | double | true | false |

注：关键字是保留字，并且不区分大小写。

1. 语言的专用符号20个：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| { | } | [ | ] | ; |
| , | + | - | / | \* |
| % | >> | << | = | & |
| | | ~ | ^ | : | :: |

1. 标示符ID、整数INTEGER、浮点数FLOATING\_PT、单字符CHAR、字符串STRING和BOOLEAN等词法规则通过下列正则表达式定义：

LETTER = [a-z] | [A- Z]

DIGIT = [0-9]

UNDERLINE= \_

ID = LETTER (UNDERLINE?( LETTER | DIGIT))\*

INTEGER\_TYPE\_SUFFIX = l | L

INTEGER = (0 | [1-9] [0-9]\*) INTEGER\_TYPE\_SUFFIX?

EXPONENT=( e | E) ( + | - )? [0-9]+

FLOAT\_TYPE\_SUFFIX= f | F | d | D

FLOATING\_PT= [0-9]+ . [0-9]\* EXPONENT? FLOAT\_TYPE\_SUFFIX?

| . [0-9]+ EXPONENT? FLOAT\_TYPE\_SUFFIX?

| [0-9]+ EXPONENT FLOAT\_TYPE\_SUFFIX?

| [0-9]+ EXPONENT? FLOAT\_TYPE\_SUFFIX

ESCAPE\_SEQUENCE = \ ( b | t | n | f | r | " | ' | \ )

CHAR= '(ESCAPE\_SEQUENCE | (~\ | ~') )'

STRING = " (ESCAPE\_SEQUENCE | (~\ | ~") )\* "

BOOLEAN = TRUE | true | FALSE | false

其中，下表给出正规表达式中运算符的含义：

|  |  |
| --- | --- |
| ~\ | ~" | 表示匹配除了 \ 和 " 外的所有字符都是合法的 |
| [a-z] | 与课件ppt中含义一致 |
| ? | 与课件ppt中含义一致 |
| + | 与课件ppt中含义一致 |
| \* | 与课件ppt中含义一致 |
| a | b | 与课件ppt中含义一致 |
| ( ) | 与课件ppt中含义一致 |

## MIDL的语法规则

MIDL的语法规则描述如下：

*specification* -> *definition* { *definition*}

*definiton* -> *type\_decl*“**;**”| *module* “**;**”

*module* -> “**module**”**ID** “**{**” definition { definition } “**}**”

*type\_decl* -> *struct\_type* | “**struct**” **ID**

*struct\_type*->**“struct”** **ID “{”** *member\_list* **“}”**

*member\_list*-> { *type\_spec* *declarators* **“;”** }

*type\_spec* -> *scoped\_name* | *base\_type\_spec* | *struct\_type*

*scoped\_name* -> [**“::”**] ID {**“::”** ID}

*base\_type\_spec*->*floating\_pt\_type*|*integer\_type*|**“char”**|**“string”**|

**“boolean”**

*floating\_pt\_type* -> **“float”** | **“double”** | **“long double”**

*integer\_type* -> *signed\_int* | *unsigned\_int*

*signed\_int*->(**“short”**|**“int16”**)

|(**“long”**|**“int32”**)

|(**“long”** **“long”**|**“int64”**)

|**“int8”**

*unsigned\_int* -> (**“unsigned”“short”**| **“uint16”**)

| (**“unsigned”“long”**| **“uint32”**)

| (**“unsigned” “long” “long”** | **“uint64”**)

| **“uint8”**

*declarators* -> *declarator* {**“,”** *declarator* }

*declarator* -> *simple\_declarator* | *array\_declarator*

*simple\_declarator* -> **ID** [**“=”** *or\_expr*]

*array\_declarator* -> **ID** “[” *or\_expr* “]” [**“=”** *exp\_list* ]

*exp\_list* -> **“[”** *or\_expr* { **“,”***or\_expr* } **“]”**

*or\_expr* -> *xor\_expr* {**“|”** *xor\_expr* }

*xor\_expr* -> *and\_expr* {**“^”** *and\_expr* }

*and\_expr* -> *shift\_expr* {**“&”***shift\_expr* }

*shift\_expr* -> *add\_expr* { (**“>>”** | **“<<”**) *add\_expr* }

*add\_expr* -> *mult\_expr* { (**“+”** | **“-”**) *mult\_expr* }

*mult\_expr* -> *unary\_expr* { (**“\*”** |**“/”**|**“%”**) *unary\_expr* }

*unary\_expr* -> [**“-”**| **“+”** | **“~”**] *literal*

*literal* -> **INTEGER** | **FLOATING\_PT** | **CHAR** | **STRING** | **BOOLEAN**

**其中，部分注意事项如下：**

1. 红色双引号表示匹配字符串本身
2. 红色{ }表示括起来的内容可以重复0至n次
3. 红色[ ]表示括起来的内容可选，即存在不存在都合法

## MIDLCompiler的词法和词法分析的任务

可以手工撰写或利用工具生成词法和语法分析程序。下面以利用Antlr4开源工具为例，给出MIDLCompiler的词法和词法分析的任务：

1. 根据Antlr4开源工具简介内容，自行搜索学习Antlr4的使用方法，主要涉及
2. 如何在G4文件中定义词法语法的规则。
3. 如何使用Antlr4工具解析G4文件，以生成相应的词法和语法分析程序。

（3）如何调用词法，语法分析程序中的方法生成对应MIDL源代码的抽象语法树。

1. 完成MIDL词法语法规则的G4文件定义。
2. 使用Antlr4编译G4文件生成MIDL语言的词法和语法分析程序的源代码。
3. 根据MIDL语法规则的文法设计抽象语法树结构，提交相应的抽象语法树定义的文件，抽象语法树的定义尽量简洁易懂，必要时提供说明。

例如：根据MIDL语法规则的文法给出相应的抽象语法树结构，这里的抽象语法树结构是自己设计的，因为语法规则文法对应的是语法分析树，需要经过凝练简化成为抽象语法树。下面是一个例子：

*stmt-sequence→statement{ ；statement }*

statement

statement

statement

……

1. 使用生成的词法和语法分析程序构建一个从MIDL源代码到抽象语法树的分析程序，并输出格式化的抽象语法树到SyntaxOut.txt文件中。

例如，针对第4条中设计的抽象语法树，使用Antlr4生成的词法语法分析程序，来构建一个由MIDL源码到抽象语法树的编译器前端。最终呈现是将构建的抽象语法树以树的格式化形式输出到SyntaxOut.txt文件。树的格式化形式可以用缩进来表示父子级关系。例如：

stmt-sequence

statement

statement

statement

...

1. 给出编译器MIDLCompiler词法和语法分析的测试方案描述，提交readme.doc，其它需要说明的问题也写在readme.doc中。测试用例需要自己设计构建，测试的目标有两个，① 测试G4中词法，文法是否正确定义。注意antlr4没有报错，只能说明语法没有问题，不代表MIDL规则定义就正确。② 测试你设计的抽象语法树是否正确构建，每个测试用例对应一个SyntaxOut.txt。测试说明写到readme.doc，还有其它需要说明的问题须也写在readme.doc中。
2. MIDLCompiler的编程语言不限制，目前Antlr4支持的语言有Java, C#, Python2|3, JavaScript, Go, C++, Swift, Dart, PHP。

## **三、任务二：**MIDLCompiler语义分析与代码生成

### 1、MIDL语言的语义分析需求

本次实验语义分析的内容有以下3条：

1. 命名冲突。同一命名空间内，不能出现相同名字的接口定义。

|  |  |
| --- | --- |
| 示例 | 语义错误 |
| struct A{  short num;  long num;  }; | 同一个struct空间下，不能有同名变量。 |
| module space{  struct A{  short a;  };  struct B{  short a;  };  }; | 同一个module下，不同的struct可以有同名变量。 |
| module space{  struct A{  short a;  };  struct A{  short a;  };  }; | 同一个module下，不能出现同名的struct。 |

1. 未定义即使用。struct结构需要先定义才能使用。

|  |  |
| --- | --- |
| 示例 | 语义错误 |
| struct A{  short a;  B b;  }; | B结构应该先定义才能引用类型 |
| module space1{  struct B{  ...  };  };  module space2{  struct A{  short a;  B b;  };  }; | 虽然B结构定义了，但是命名空间的引用不对。应该是 space1::B |

1. 字面量类型检查。字面量的数据类型需要和变量类型相同或兼容。

|  |  |
| --- | --- |
| 示例 | 语义错误 |
| struct A{  short a=’a’;  }; | a是整型变量，字面量却是字符类型 |
| struct A{  short a=100000;  }; | short为有符号短整型，最大值不超过 |
| struct A{  short a=15.24;  }; | a是整型变量，字面量却是浮点类型 |
| struct A{  short a[4]=[10,12,45.34,’a’];  }; | a是整型数组，数组字面量里必须保证数据类型的统一 |

### 2、MIDL语言的C++ 代码生成要求

MIDL作为接口定义语言，具有跨平台的好处。在分布式应用中，我们可以使用IDL编译器将IDL接口定义转换为特定编程语言实现的客户端或服务端桩程序。例如，商业IDL编译器rtiddsgen工具，针对特定的IDL（struct.idl）会生成6个实现文件，分别是实现文件（struct.h/struct.cxx），Plugin文件（structPlugin.h/structPlugin.cxx），Support文件（structSupport.h/structSupport.cxx）。

本次实验代码生成部分为了降低难度，仅需要生成一份实现文件即可。输入为MIDL源码，输出为一份.hxx实现文件。相关测试用例详见“代码生成用例集.zip”，可对照测试用例进行代码生成的逻辑编写。

### 3、MIDLCompiler的语义和代码生成任务

在实验一内容的基础上，完成语义分析，实现能够对上文所提的3条语义错误进行检查和报错。报错提示要给出错误位置，以及错误类型。并给出测试说明文档。

从抽象语法树生成对应的C++代码，测试用例参考“代码生成用例集.zip”，依据用例集中的输入输出，总结抽象语法节点的生成规则，并编写代码生成模块。（代码生成可使用ppt中第6章代码生成的方法，也可以采用附录A中介绍的StringTemplate模板方式生成目标代码，并配有学习视频“StringTemplate模板语法介绍.mp4”和“StringTemplate模板语法介绍.mp4”）。

针对语义分析和代码生成设计测试用例，给出测试方法描述，提交readme.doc，如果你还有其他需要说明的问题须写在readme.doc中。

整合实验一实验二内容，构建一个完整的MIDL To C++的源到源的编译器，并依据实践内容撰写一篇详细实验报告，报告内容包括但不限于实验环境，实验内容，实验流程，测试说明，将这两次实验课中的设计，实现，测试等内容讲述清楚即可。

附录A:

# 四、Antlr4开源工具简介

Antlr4（Another Tool for Language Recognition）是一款基于Java开发的开源的语法分析器生成工具，能够根据语法规则文件生成对应的语法分析器，广泛应用于DSL构建，语言词法语法解析等领域。Antlr可以生成不同target的AST，包括Java、C++、JS、Python、C#等，可以满足不同语言的开发需求。下面是Antlr4相关网址：

官方网址： <https://www.antlr.org/>

Github开源地址：<https://github.com/antlr/antlr4>

官方文档：<https://github.com/antlr/antlr4/blob/master/doc/index.md>

关于如何使用antlr4生成词法，语法分析程序，有两种方法。

首先，确保你在工程引入了org.antlr.v4.runtime 运行时库。

如何构建开发环境 ---> <https://github.com/antlr/antlr4/blob/master/doc/java-target.md>

继承Listener来实现对每类节点的监听，查阅以下链接学习

<https://media.pragprog.com/titles/tpantlr2/listener.pdf>

这种方法的好处是用户无需自己遍历分析树，通过写监听函数即可实现对每类节点的监听。缺点是如果每类节点都有通用的操作，那么用户不得不为每类节点写监听函数，代码比较臃肿。

自己遍历分析树。下面提供了一段Java示例程序供参考学习。这种方法的好处与缺点和第一种刚好相反。自由度更高，自行遍历可以获取到每个节点的属性，并进行一些通用的操作，代码会更精简。缺点是需要自己写递归函数来遍历分析树，对自身算法水平有要求。

|  |
| --- |
| IDLLexer idlL = null;  IDLParser parser = null;  CharStream input = CharStreams.fromString(Util.readFile(idlPath));  idlL = new IDLLexer(input);  CommonTokenStream tokens = new CommonTokenStream(idlL);  parser = new IDLParser(tokens);  ParseTree tree = parser.specification();  if (parser.getNumberOfSyntaxErrors()>0)  throw new SyntaxException(String.format("有 %s 处语法错误，请根据报错信息进行修改后再次尝试 !",parser.getNumberOfSyntaxErrors()));  /\* 渲染所有的stg模板并生成 \*/  context = buildIDLTree(tree,parser);  for (STGroup stg:  stgs) {  targetCodes.add(\_generateTargetCode(context,stg,semanticStg));  }  return targetCodes; |
| public static IDLTreeNode buildIDLTree(ParseTree t,Parser parser) throws NullPointerException{  /\* 错误节点 \*/  if ( t instanceof ErrorNode) {  return null;  }  /\* 终结符节点 \*/  else if ( t instanceof TerminalNode) {  String tokenType = tokensTypeCode.get(((TerminalNode) t).getSymbol().getType());  IDLTreeNode tNode = new IDLTreeNode(tokenType,"T",Trees.getNodeText(t,parser),null,null);  /\* 加入到上下文环境中去 \*/  tNode.setHashCode(tNode.hashCode());  contextMap.put(tNode.hashCode(),tNode);  return tNode;  }  /\* 非终结符节点 \*/  IDLTreeNode root = new IDLTreeNode(Trees.getNodeText(t,parser),"NONT",null,null,null);  root.setHashCode(root.hashCode());  /\* 加入到上下文环境中去 \*/  contextMap.put(root.hashCode(),root);  ArrayList<IDLTreeNode> childs = new ArrayList<IDLTreeNode>();  /\* 对分析树递归 \*/  RuleNode r = (RuleNode)t;  int n = r.getChildCount();  for (int i = 0; i<n; i++) {  ParseTree rc = r.getChild(i);  IDLTreeNode child = buildIDLTree(rc,parser);  childs.add(child);  /\* 为孩子节点设置父亲节点 \*/  child.setPnode(root);  }  /\* 为当前节点设置孩子节点列表 \*/  root.setNodes(childs);  return root;  } |

其中对于tokensTypeCode存的是antlr4生成的IDL.tokens 文件中的token与token id的字典。

# 五、StringTemplate模板引擎介绍

StringTemplate（简称ST）是一个基于Java的模板引擎库（同时支持C#、Python、Ruby），可以用于生成源代码、Web页面、电子邮件等各种有格式的文本。StringTemplate不同于其他类似的模板引擎的显著特点是严格执行模型视图分离，尤其擅长多目标代码生成、多种网站风格管理以及网站多语言版本生成等。目前支持的语言有Java,C#,JavaScript(not stable),Python和Objective-C。

在我们提供的“代码生成用例集.zip”中，可以发现生成的hxx文件是以特定的格式进行组织的，且有大量固定的字符串。如果我们使用完全的字符串拼接，有两个坏处：

（1）可读性差。因为是在源码中进行字符串拼接，因此代码的整洁度和可读性都不好。

（2）可维护性差。同样因为是在源码中进行字符串拼接，所以在字符串格式控制上难以维护，可能稍有不慎就导致比较大的格式问题。

StringTemplate模板引擎可以很好的解决这个问题。用户可以通过定义STG文件，定义模板函数来将视图与模型分离（view-model)，进而直接控制字符文本的格式，所见即所得。而模板中变化的部分可以通过函数参数的形式传入模板中进行渲染。相关文档如下链接所示：

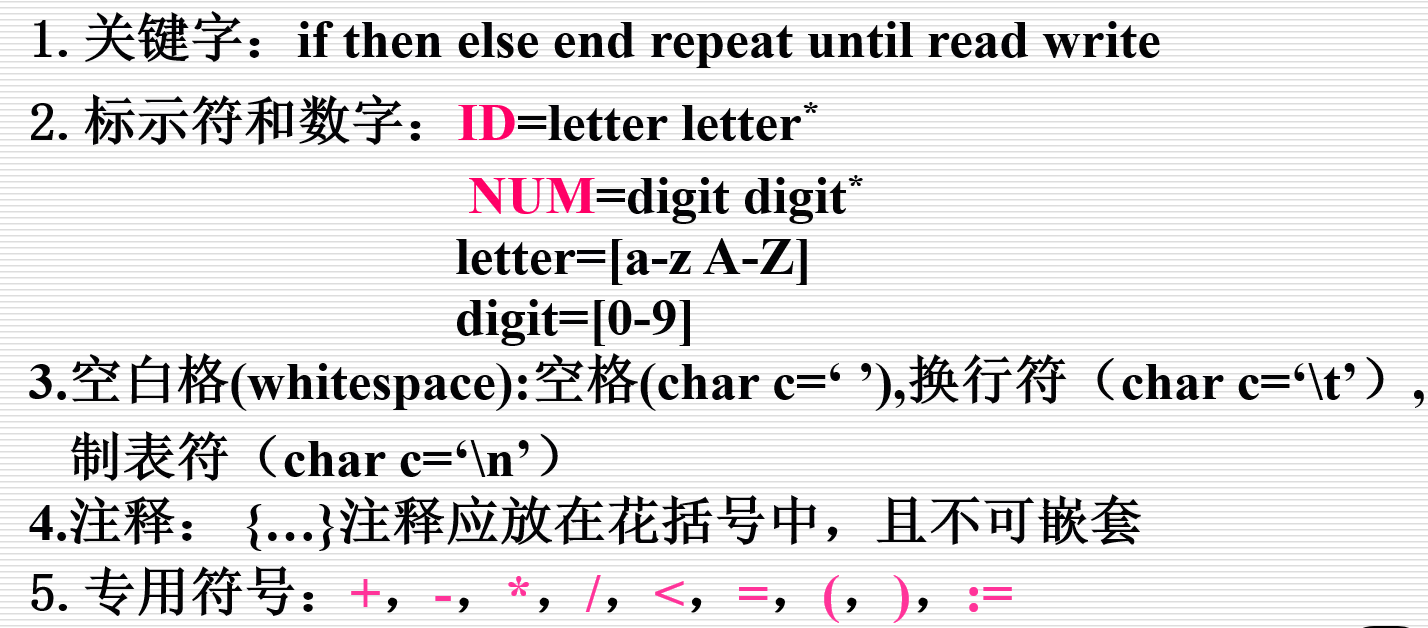
<https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>

[](StringTemplate模板语法介绍.mp4)

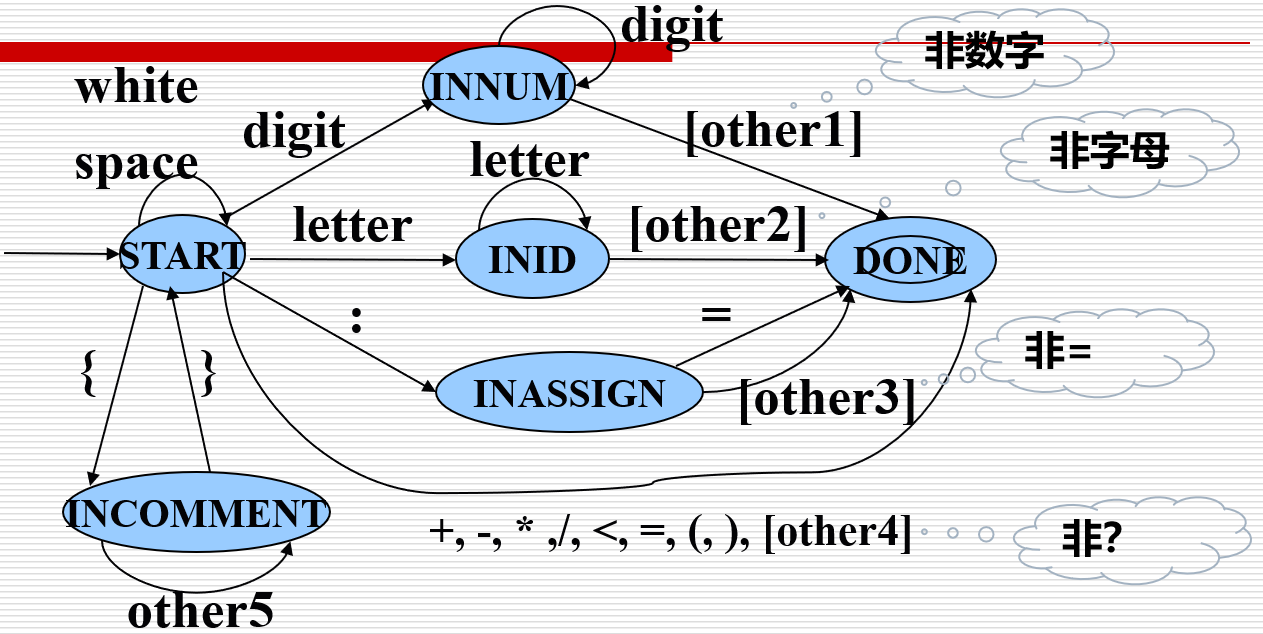
[](JAVA中调用ST模板引擎演示视频.mp4)

# 六、手工撰写词法分析程序参考案例

## 1、Tiny语言的词法规则定义



Tiny语言的词法规则对应的确定性有穷自动机模型



## 2、基于自动机模型的Tiny语言的词法程序

Scan.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: scan.h \*/

/\* The scanner interface for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_SCAN\_H\_

#define \_SCAN\_H\_

/\* MAXTOKENLEN is the maximum size of a token \*/

#define MAXTOKENLEN 40

/\* tokenString array stores the lexeme of each token \*/

extern char tokenString[MAXTOKENLEN+1];

/\* function getToken returns the

\* next token in source file

\*/

TokenType getToken(void);

#endif

Scan.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: scan.c \*/

/\* The scanner implementation for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "globals.h"

#include "util.h"

#include "scan.h"

/\* states in scanner DFA \*/

typedef enum

{ START,INASSIGN,INCOMMENT,INNUM,INID,DONE }

StateType;

/\* lexeme of identifier or reserved word \*/

char tokenString[MAXTOKENLEN+1];

/\* BUFLEN = length of the input buffer for

source code lines \*/

#define BUFLEN 256

static char lineBuf[BUFLEN]; /\* holds the current line \*/

static int linepos = 0; /\* current position in LineBuf \*/

static int bufsize = 0; /\* current size of buffer string \*/

static int EOF\_flag = FALSE; /\* corrects ungetNextChar behavior on EOF \*/

/\* getNextChar fetches the next non-blank character

from lineBuf, reading in a new line if lineBuf is

exhausted \*/

static int getNextChar(void)

{ if (!(linepos < bufsize))

{ lineno++;

if (fgets(lineBuf,BUFLEN-1,source))

{ if (EchoSource) fprintf(listing,"%4d: %s",lineno,lineBuf);

bufsize = strlen(lineBuf);

linepos = 0;

return lineBuf[linepos++];

}

else

{ EOF\_flag = TRUE;

return EOF;

}

}

else return lineBuf[linepos++];

}

/\* ungetNextChar backtracks one character

in lineBuf \*/

static void ungetNextChar(void)

{ if (!EOF\_flag) linepos-- ;}

/\* lookup table of reserved words \*/

static struct

{ char\* str;

TokenType tok;

} reservedWords[MAXRESERVED]

= {{"if",IF},{"then",THEN},{"else",ELSE},{"end",END},

{"repeat",REPEAT},{"until",UNTIL},{"read",READ},

{"write",WRITE}};

/\* lookup an identifier to see if it is a reserved word \*/

/\* uses linear search \*/

static TokenType reservedLookup (char \* s)

{ int i;

for (i=0;i<MAXRESERVED;i++)

if (!strcmp(s,reservedWords[i].str))

return reservedWords[i].tok;

return ID;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* the primary function of the scanner \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* function getToken returns the

\* next token in source file

\*/

TokenType getToken(void)

{ /\* index for storing into tokenString \*/

int tokenStringIndex = 0;

/\* holds current token to be returned \*/

TokenType currentToken;

/\* current state - always begins at START \*/

StateType state = START;

/\* flag to indicate save to tokenString \*/

int save;

while (state != DONE)

{ int c = getNextChar();

save = TRUE;

switch (state)

{ case START:

if (isdigit(c))

state = INNUM;

else if (isalpha(c))

state = INID;

else if (c == ':')

state = INASSIGN;

else if ((c == ' ') || (c == '\t') || (c == '\n'))

save = FALSE;

else if (c == '{')

{ save = FALSE;

state = INCOMMENT;

}

else

{ state = DONE;

switch (c)

{ case EOF:

save = FALSE;

currentToken = ENDFILE;

break;

case '=':

currentToken = EQ;

break;

case '<':

currentToken = LT;

break;

case '+':

currentToken = PLUS;

break;

case '-':

currentToken = MINUS;

break;

case '\*':

currentToken = TIMES;

break;

case '/':

currentToken = OVER;

break;

case '(':

currentToken = LPAREN;

break;

case ')':

currentToken = RPAREN;

break;

case ';':

currentToken = SEMI;

break;

default:

currentToken = ERROR;

break;

}

}

break;

case INCOMMENT:

save = FALSE;

if (c == EOF)

{ state = DONE;

currentToken = ENDFILE;

}

else if (c == '}') state = START;

break;

case INASSIGN:

state = DONE;

if (c == '=')

currentToken = ASSIGN;

else

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

currentToken = ERROR;

}

break;

case INNUM:

if (!isdigit(c))

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

state = DONE;

currentToken = NUM;

}

break;

case INID:

if (!isalpha(c))

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

state = DONE;

currentToken = ID;

}

break;

case DONE:

default: /\* should never happen \*/

fprintf(listing,"Scanner Bug: state= %d\n",state);

state = DONE;

currentToken = ERROR;

break;

}

if ((save) && (tokenStringIndex <= MAXTOKENLEN))

tokenString[tokenStringIndex++] = (char) c;

if (state == DONE)

{ tokenString[tokenStringIndex] = '\0';

if (currentToken == ID)

currentToken = reservedLookup(tokenString);

}

}

if (TraceScan) {

fprintf(listing,"\t%d: ",lineno);

printToken(currentToken,tokenString);

}

return currentToken;

} /\* end getToken \*/

Global.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: globals.h \*/

/\* Global types and vars for TINY compiler \*/

/\* must come before other include files \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_GLOBALS\_H\_

#define \_GLOBALS\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#ifndef FALSE

#define FALSE 0

#endif

#ifndef TRUE

#define TRUE 1

#endif

/\* MAXRESERVED = the number of reserved words \*/

#define MAXRESERVED 8

typedef enum

/\* book-keeping tokens \*/

{ENDFILE,ERROR,

/\* reserved words \*/

IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,

/\* multicharacter tokens \*/

ID,NUM,

/\* special symbols \*/

ASSIGN,EQ,LT,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI

} TokenType;

extern FILE\* source; /\* source code text file \*/

extern FILE\* listing; /\* listing output text file \*/

extern FILE\* code; /\* code text file for TM simulator \*/

extern int lineno; /\* source line number for listing \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\* Syntax tree for parsing \*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef enum {StmtK,ExpK} NodeKind;

typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;

typedef enum {OpK,ConstK,IdK} ExpKind;

/\* ExpType is used for type checking \*/

typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode

{ struct treeNode \* child[MAXCHILDREN];

struct treeNode \* sibling;

int lineno;

NodeKind nodekind;

union { StmtKind stmt; ExpKind exp;} kind;

union { TokenType op;

int val;

char \* name; } attr;

ExpType type; /\* for type checking of exps \*/

} TreeNode;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\* Flags for tracing \*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* EchoSource = TRUE causes the source program to

\* be echoed to the listing file with line numbers

\* during parsing

\*/

extern int EchoSource;

/\* TraceScan = TRUE causes token information to be

\* printed to the listing file as each token is

\* recognized by the scanner

\*/

extern int TraceScan;

/\* TraceParse = TRUE causes the syntax tree to be

\* printed to the listing file in linearized form

\* (using indents for children)

\*/

extern int TraceParse;

/\* TraceAnalyze = TRUE causes symbol table inserts

\* and lookups to be reported to the listing file

\*/

extern int TraceAnalyze;

/\* TraceCode = TRUE causes comments to be written

\* to the TM code file as code is generated

\*/

extern int TraceCode;

/\* Error = TRUE prevents further passes if an error occurs \*/

extern int Error;

#endif

Util.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: util.h \*/

/\* Utility functions for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_UTIL\_H\_

#define \_UTIL\_H\_

/\* Procedure printToken prints a token

\* and its lexeme to the listing file

\*/

void printToken( TokenType, const char\* );

/\* Function newStmtNode creates a new statement

\* node for syntax tree construction

\*/

TreeNode \* newStmtNode(StmtKind);

/\* Function newExpNode creates a new expression

\* node for syntax tree construction

\*/

TreeNode \* newExpNode(ExpKind);

/\* Function copyString allocates and makes a new

\* copy of an existing string

\*/

char \* copyString( char \* );

/\* procedure printTree prints a syntax tree to the

\* listing file using indentation to indicate subtrees

\*/

void printTree( TreeNode \* );

#endif

# 七、手工撰写语法分析程序参考案例

## 1、Tiny语言语法规则定义

*program -> stmt-sequence*

*stmt-sequence ->  stmt-sequence;statement|statement*

*statement -> if-stmt|repeat-stmt;assign-stmt;read-stmt;write-stmt*

*if-stmt -> if exp then stmt- sequence end*

*| if exp then stmt- sequence else stmt-sequence end*

*repeat-stmt -> repeat stmt-sequence until exp*

*assign-stmt -> identifier := exp*

*read-stmt -> read identifier*

*write-stmt -> write exp*

*exp -> simple-exp comparison-op simple-exp | simple-exp*

*comparison-op -> <| =*

*simple-exp -> simple-exp addop term |term*

*addop -> +|-*

*term -> term mulop factor |factor*

*mulop -> \*|/*

*factor -> (exp)|number | identifier*

## 2、语法分析程序实现步骤

### 2.1将文法用EBNF表示

*program*→*stmt-sequence*

*stmt-sequence*→*statement*{ **；***statement* }

*statement*→*if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt*

*if-stmt*→**if** *exp* **then** *stmt-sequence* **[ else** *stmt-squence* **] end**

*repeat-stmt*→**repeat** *stmt-sequence* **until** *exp*

*assign-stmt*→**identifier :=** *exp*

*read-stmt*→**read** **identifier**

*write-stmt*→**write** *exp*

*exp*→*simple-exp* **[** *comparison-op simple-exp* **]**

*comparison-op*→**<** | **=**

*simple-exp*→*term* { *addop term* }

*addop*→ **+**| **-**

*term*→*factor* { *mulop factor* }

*mulop*→ **\*** | **/**

*factor*→**(** *exp* **)** | **number** |**identifier**

### 2.2 抽象语法树设计

*program*→*stmt-sequence*

stmt-sequence

*stmt-sequence→statement{ ；statement }*

statement

statement

statement

……

*if-stmt→if exp then stmt-sequence [ else stmt-squence ] end*

If

exp

stmt-sequence

stmt-sequence

*repeat-stmt*→**repeat** *stmt-sequence* **until** *exp*

*repeat*

*stmt-sequence*

*exp*

*assign-stmt*→**identifier :=** *exp*

*Assign（identifier）*

*exp*

*read-stmt*→**read** **identifier**

*read（identifier）*

*exp*

*write-stmt*→**write** *exp*

*write*

*exp*

*exp→simple-exp [ comparison-op simple-exp ]*

*comparison-op→****<*** *|* ***=***

*comparison-op(****<*** *|* ***=)***

*simple-exp*

*simple-exp*

*simple-exp→term { addop term }*

*addop→* ***+****|* ***-***

*Term*

*op(+|-)*

*(+|-*

*Term*

……

*term→factor { mulop factor }*

*mulop→* ***\**** *|* ***/***

*factor*

*mulop(\*|/)*

*(+|-*

*factor*

……

*factor*→**(** *exp* **)** | **number** |**identifier**

*exp*| **number** |**identifier**

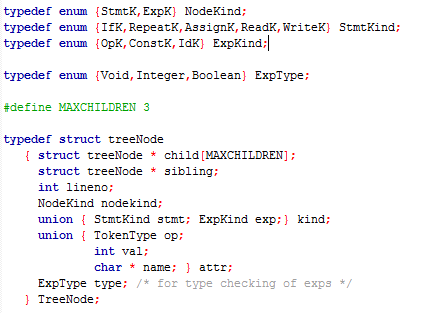
### 2.3定义数据结构

TINY有两种基本的结构类型：语句和表达式。

语句共有5类：（if语句、repeat语句、assign语句、read语句和read语句），

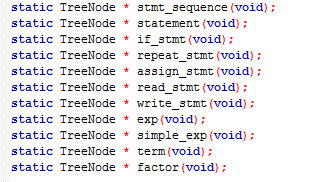
表达式共有3类（算符表达式、常量表达式和标识符表达式）。

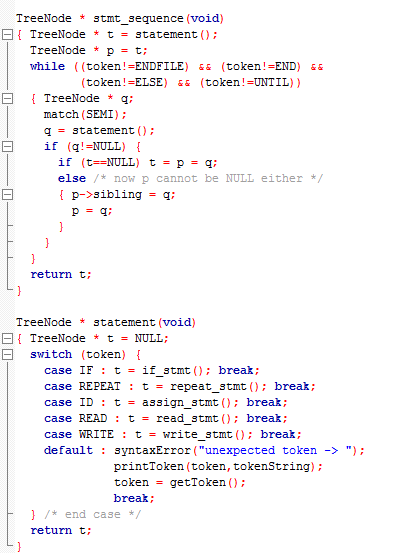
以C代码为例：

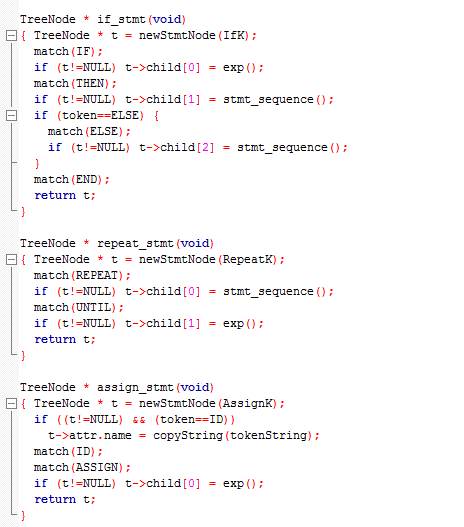


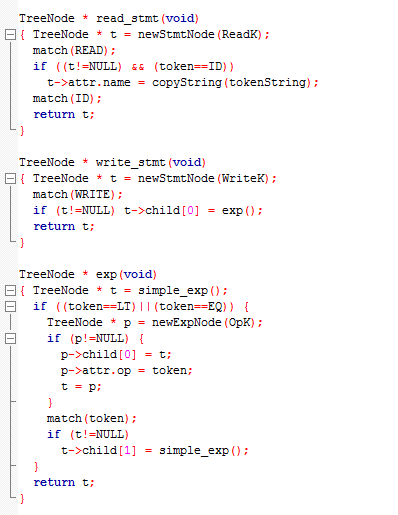
### 2.4实现递归下降分析程序

1、语法分析递归下降核心代码









*simple-exp*→*term* { *addop term* }*3+4-5*

