编译 课程项目报告

王盛业　白彦博

10300720116
10300240026

2013 年 6 月 19 日

综述

本次课程项目中，我们完成了 Tiger 编译器的前端部分，即从源代码输入到生成抽象语法树。测试表明，我们能够分析语法正确的 Tiger 程序，并生成抽象语法树以便后续处理。

我们选用了使用 LALR(1)文法的 flex 和 bison 套件来作为我们的辅助工具，并且使用 C++ 的面向对象设计方法组织数据结构。这使得我们的分析简单而高效。

我们完成了一部分的错误提示和错误修复。flex 和 bison 的一些技巧性用法让我们能够在词法和语法分析的同时记录错误信息。我们能够直观地提示出现的错误。对于几种简单的错误，我们能够尝试分析并继续执行。

限于时间，我们没能来得及进一步从语义的层面上进行分析。然而清晰的抽象语法树表示可以让我们很方便地扩充这方面的功能。若能有更多时间，我们将会完成语义分析并尝试解释执行 Tiger 代码。

我们共完成了两个工具，分别完成以下功能:

1. 从用户输入读入设计，进行词法和句法的分析，建立抽象语法树，在理解语法的基础上将树形表示转换为代码重新输出。这个目标让我们在编码的过程中更轻松的调试。另外，这个工作也可以作为“Tiger 智能编辑器”的一部分。即编辑器能够理解语义，给出用户编码过程中词法和语法方面的建议，提高工作效率。

2. 从用户输入读入设计，进行词法和句法的分析，建立抽象语法树，将树形表示用 dot 语言描述并输出。而后 dot 语言描述可以经过 Graphviz 工具转换为表达清晰的图形。这个目标是我们对设计要求的满足，即我们能直观的看出抽象语法树在内存中的表示。

以上两套工具使用相同的后端数据结构，仅包含主函数的文件内容不同。这说明，我们的抽象语法树数据结构设计合理，能够满足不同条件下的需求。

我们提交的文件包括以上两个工具对所有 51 个测试用例运行的结果。经过观察，它们的结果均正确。

词法部分

我们使用了 flex 来分析词法。flex 是基于正则表达式的词法分析工具生成器。在词法分析部分，我们做了以下几项工作:

1. 过滤注释。按照 Tiger 语言的要求，我们的词法分析部分支持注释的嵌套。

2. 关键字解释。对于 Tiger 语言规定关键字和标点符号，我们返回不同的标志(token)。

3. 字符串转义。我们按照 C 语言的风格转义字符串。例如\n 被解释为换行，\x65 被解释为字母'A'。目前我们能够支持八进制、十六进制和常见的转义符。我们

设置了转义开关，在进行实际分析的时候打开，进行展示（如输出树形和代码格式化）则不需要打开。

4. 行数和列数统计。我们使用了 flex 语法分析器的一些技巧来进行行数和列数统计，以方便错误提示。

对应的词法规则文件名是`Lexicon.l`。正则式描述部分摘录如下：

```
ID              [a-zA-Z][0-9a-zA-z_]*
BLANK           [ \t\n\r\f]
OCTCH           \\[0-7]{3}
HEXCH           \\x[0-9a-fA-F]{2}

%x              FIRSTLINE
%x              COMMENT
%x              STRING
```

```
<FIRSTLINE>.*               {   ...; BEGIN INITIAL; yyless(0);  }
<INITIAL,COMMENT,STRING>\n.*{   ...; yyless(1);                 }
"array"                     {   return  TOK_ARRAY;              }
"break"                     {   return  TOK_BREAK;              }
"do"                        {   return  TOK_DO;                 }
"else"                      {   return  TOK_ELSE;               }
"end"                       {   return  TOK_END;                }
"for"                       {   return  TOK_FOR;                }
"function"                  {   return  TOK_FUNCTION;           }
"if"                        {   return  TOK_IF;                 }
"in"                        {   return  TOK_IN;                 }
"let"                       {   return  TOK_LET;                }
"nil"                       {   return  TOK_NIL;                }
"of"                        {   return  TOK_OF;                 }
"then"                      {   return  TOK_THEN;               }
"to"                        {   return  TOK_TO;                 }
"type"                      {   return  TOK_TYPE;               }
"var"                       {   return  TOK_VAR;                }
"while"                     {   return  TOK_WHILE;              }
"+"                         {   return  TOK_PLUS_SIGN;          }
"-"                         {   return  TOK_MINUS_SIGN;         }
"*"                         {   return  TOK_MULT_SIGN;          }
"/"                         {   return  TOK_DIV_SIGN;           }
"&"                         {   return  TOK_AND;                }
"|"                         {   return  TOK_OR;                 }
```

```
"="                     {   return  TOK_EQUALS;             }
"<>"                    {   return  TOK_NEQ;                }
"<"                     {   return  TOK_LT;                 }
"<="                    {   return  TOK_LTE;                }
">"                     {   return  TOK_GT;                 }
">="                    {   return  TOK_GTE;                }
":="                    {   return  TOK_ASSIGN;             }
";"                     {   return  TOK_SEMIC;              }
","                     {   return  TOK_COMMA;              }
":"                     {   return  TOK_COLON;              }
"."                     {   return  TOK_DOT;                }
"("                     {   return  TOK_LBR;                }
")"                     {   return  TOK_RBR;                }
"["                     {   return  TOK_LSQB;               }
"]"                     {   return  TOK_RSQB;               }
"{"                     {   return  TOK_LCURLB;             }
"}"                     {   return  TOK_RCURLB;             }
[0-9]+                  {   ...; return  TOK_INTEGER;       }
{ID}                    {   ...; return  TOK_ID;            }
{BLANK}                 {   /* ignore */                    }
"/*"                    {   BEGIN COMMENT; ++ cmntNest;     }
<COMMENT>"/*"           {   ++ cmntNest;                    }
<COMMENT>"*/"           {   if (!--cmntNest) BEGIN INITIAL; }
<COMMENT>.              |
<COMMENT>"\n"           {   /* do nothing */                }
"\""                    {   BEGIN STRING;                   }

<STRING>"\""            {   BEGIN INITIAL;return TOK_STRING;}
<STRING>"\\a"           {   ...;                            }
<STRING>"\\b"           {   ...;                            }
<STRING>"\\f"           {   ...;                            }
<STRING>"\\n"           {   ...;                            }
<STRING>"\\r"           {   ...;                            }
<STRING>"\\t"           {   ...;                            }
<STRING>"\\v"           {   ...;                            }
<STRING>"\\'"           {   ...;                            }
<STRING>"\\\""          {   ...;                            }
<STRING>"\\\\"          {   ...;                            }
<STRING>"\\?"           {   ...;                            }
<STRING>{OCTCH}         {   ...;                            }
<STRING>{HEXCH}         {   ...;                            }
```

```
<STRING>.                    {   ...;                    }
```

词法部分参考自 Cormac Redmond 的主页。我们利用了其中的正则式描述和标记命名。来源：`http://www.credmond.net/projects/tiger-lexical-analyser/`

行数和列数统计原理

我们使用了一条特殊的规则，即`\n.*`来完成对每一行的匹配。对于它的规则，我们对行数记号加一、清零列号，再使用`yyless(1)`将除了换行符以外的字符交给词法分析器继续处理。

另外我们使用了`YY_USER_ACTION`宏来统计列数。具体用法参见词法描述文件。

语法分析

我们使用了 bison 作为语法分析器产生工具。它使用 LALR(1)文法。他的好处是语法描述方便而清晰，不需要像 LL(1)文法那样改写。而偏好左递归描述让我们更容易处理表达式组这样的元素。

我们参考了 Cormac Redmond 主页上关于 Tiger 语法分析器的描述。在他的基础上，我们做了更多的修改：

1. 我们将他的文法修改地更加直观。Cormac Redmond 给出的非 LL(1)文法经过了消除左递归。由于我们使用 LALR(1)文法分析器，我们没有必要使用满足 LL(1)但可读性较差的文法。因而我们把文法修改得更加直观，去除了冗余的 Pr 记号(Cormac Redmond 用 Pr 来表示 Prime)。这样有助于 bison 进行分析。因为 bison 将不会再需要先规约派生规则去匹配主规则。同时，左递归让我们更容易处理用记号隔开的相同元素。

2. 我们增加了优先级设定。通过设定优先级规则，我们将 bison 警告的规约/移进冲突全部解决。这确保文法分析能够正确进行。为了达到这一目标，我们改写了一条规则的描述，即`ID[Exp]`规则，用这条规则的规约去匹配表达和左值对这样描述的同时要求，解决了他们的移进/规约冲突。

3. 我们增加了错误处理。对于表达式组，我们选择`;`作为同步，这样某些表达式错误的情况下仍然能够正确处理后续的表达式。我们本可以做更多的错误处理，但是过多的错误处理反而容易让用户困惑（如果从一个参数很多的函数的形参列表中删除了一项，却报告形参列表不匹配，用户是否会更加难以找到错误所在）。有理由相信用户当为自己的错误负责，工具应当提示错误，在没有把握之前过多修复错误也许并不是一个好主意。

4. 我们增加了错误提示。我们能够指出，哪一行那一列出现错误以及错误的类型。给出的错误提示通常能够帮助用户发现错误并改正。

对应的词法规则文件名是`Grammar.y`。文法规则摘抄如下:

```
Prog            :       Exp
Exp             :       LValue
                |       TOK_INTEGER
                |       TOK_NIL
                |       TOK_STRING
                |       TOK_ID TOK_LBR ArgList TOK_RBR
                |       Exp BinOp_L1 Exp
                |       Exp BinOp_L2 Exp
                |       Exp BinOp_L3 Exp
                |       Exp BinOp_L4 Exp
                |       Exp BinOp_L5 Exp
                |       UnOp Exp
                |       TOK_ID TOK_LCURLB FieldExpList TOK_RCURLB
                |       TOK_LBR ExpList TOK_RBR
                |       LValue TOK_ASSIGN Exp
                |       TOK_IF Exp TOK_THEN Exp
                |       TOK_IF Exp TOK_THEN Exp TOK_ELSE Exp
                |       TOK_WHILE Exp TOK_DO Exp
                |       TOK_FOR TOK_ID TOK_ASSIGN Exp
                                        TOK_TO Exp TOK_DO Exp
                |       TOK_BREAK
                |       TOK_LET DecList TOK_IN ExpList TOK_END
                |       IdSqB TOK_OF Exp
DecList         |       Dec
Dec             :       TyDec
                |       VarDec
                |       FuncDec
TyDec           :       TOK_TYPE TOK_ID TOK_EQUALS Ty
Ty              :       TOK_ID
                |       TOK_LCURLB FieldList TOK_RCURLB
                |       TOK_ARRAY TOK_OF TOK_ID
VarDec          :       TOK_VAR TOK_ID TOK_ASSIGN Exp
                |       TOK_VAR TOK_ID TOK_COLON TOK_ID
                                        TOK_ASSIGN Exp
FuncDec         :       TOK_FUNCTION TOK_ID TOK_LBR FieldList
                                        TOK_RBR TOK_EQUALS Exp
                |       TOK_FUNCTION TOK_ID TOK_LBR FieldList
                            TOK_RBR TOK_COLON TOK_ID TOK_EQUALS Exp
IdSqB           :       TOK_ID SqBExp
LValue          :       TOK_ID
```

```
                |       LValue TOK_DOT TOK_ID
                |       IdSqB
                |       LValue SqBExp
ExpList         :       /* nothing */
                |       Exp
                |       ExpList TOK_SEMIC Exp
                |       ExpList error TOK_SEMIC Exp
ArgList         :       /* nothing */
                |       Exp
                |       ArgList TOK_COMMA Exp
FieldList       :       /* nothing */
                |       TOK_ID TOK_COLON TOK_ID
                |       FieldList TOK_COMMA TOK_ID
                                  TOK_COLON TOK_ID
FieldExpList    :       /* nothing */
                |       TOK_ID TOK_EQUALS Exp
                |       FieldExpList TOK_COMMA TOK_ID
                                     TOK_EQUALS Exp
                |       FieldExpList error TOK_COMMA
BinOp_L1        :       TOK_MULT_SIGN
                |       TOK_DIV_SIGN
BinOp_L2        :       TOK_PLUS_SIGN
                |       TOK_MINUS_SIGN
BinOp_L3        :       TOK_NEQ
                |       TOK_LT
                |       TOK_LTE
                |       TOK_GT
                |       TOK_GTE
                |       TOK_EQUALS
BinOp_L4        :       TOK_AND
BinOp_L5        :       TOK_OR
UnOp            :       TOK_MINUS_SIGN
```

错误提示和错误修复

我们记录了行号和位置以提示错误。以代码格式化器运行结果为例，如果我们输入错误的一段代码，例如

```
(a(1, 2, 3); b(4, 5, 6); c(7, 8y, 9); d(x, y, z))
------------------------------^^-----------------
```

注意到8y是一个错误的输入，因为它不是一个合法的表达式。程序的输出如下:

```
====================Project Compiler T====================
=                A Tiger Language Formatter                =
=                           By                             =
=                Shengye Wang  &  Yanbo Bai                 =
==========================================================
Compiler-T: Start parsering.
(a(1, 2, 3); b(4, 5, 6); c(7, 8y, 9); d(x, y, z))
Parser: syntax error, unexpected identifier, expecting "," or ")".
"y" at line 1: (a(1, 2, 3); b(4, 5, 6); c(7, 8y, 9); d(x, y, z))
               ..............................^.................
Statement discarded, continuing.
Compiler-T: Parser returned 0.
Compiler-T: 20 AST nodes created.
Compiler-T: Rewriting Tiger source.
(
    a(1, 2, 3);
    b(4, 5, 6);
    d(x, y, z)
)
Compiler-T: Done. 3.080836 seconds elapsed.
==========================================================
```