

编译原理课程设计报告

PL/0 编译器设计与实现

王君卓

1152252

1 设计概述

本程序使用 Python 语言进行编写，需要安装 CPython 解释器才可以运行，程序中使用了多种 Python 的特性，与 C++、Java 等语言有所不同。是一个使用 LR(1) 分析器的编译器框架，使用方法类似 yacc。框架的核心是一个 LR 主控程序，接受 bison 兼容或 bison 生成的 xml 文件配置语法分析器、一个词法分析迭代器、以及一个语义分析器。使用不同的配置文件可以分析不同的语言。本设计使用可以分析扩展 PL/0 语言的相关文件来配置本框架。

由于 PL/0 是 Pascal 语言的一个子集，其代码应该可以在 Pascal 编译器中通过编译。不幸的是，由于 PL/0 所有的变量都是整型，不需要单独声明，而 Pascal 支持多种变量，需要声明。导致 PL/0 的代码无法在 Free Pascal 中编译，本设计支持的代码在添加变量类型后，可以通过 Free Pascal 编译器编译出可执行文件。由于本设计支持更多 PL/0 特性，包括函数调用、函数嵌套定义等，并完整的实现了基于栈的运行时空组织，拥有静态链、动态链、返回地址等基本高级语言应该拥有的功能。所以，中间代码未使用四元式，使用了 PL/0 兼容的 P-Code，更多本设计遵循的设计标准见PL0.pdf 或附录。

2 文件概述

本程序使用如下文件实现了 PL/0 编译器：

文件	功能
compiler.py	编译器文件，执行本文件编译
lrparser.py	LR 分析器主控程序，通过 xml 文件配置
scanner.py	词法分析器
syntaxer.py	语义分析器
backend.py	语义分析器后端文件，维护符号表等数据结构
translate.py	中间代码翻译器
pl0.xml	PL/0 语法配置文件，Bison 生成
pl0.y	PL/0 语法描述文件，用于生成 pl0.xml
Simulator/main.cpp	PL/0 中间代码 P-Code 解释器

表 1: 文件用途

2.1 工作流程

编译器的工作流程如下：

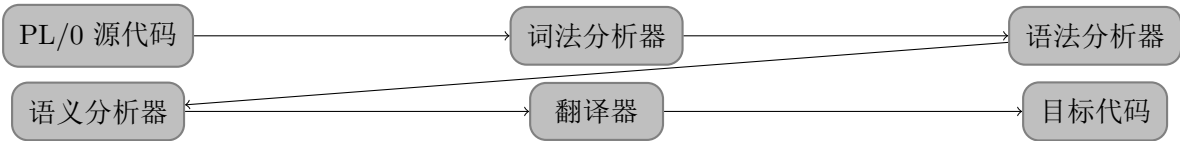


图 1: 工作流程

其中，源文件、中间代码、目标代码使用纯文本文件保存。

2.2 词法分析器

词法分析器文件名为`scanner.py`，内含 2 个类，`Scanner` 与 `Token`。其中 `Scanner` 为词法分析器，使用了 Python 自带的正则表达式模块 `re`。通过正则表达式，过滤出符合 PL/0 语言定义的单词。这些单词存放在内存中，同时 `Scanner` 的实例也是一个迭代器，以便语法分析器调用。`Token` 类保存了不同单词所在位置、原文等信息。根据约定，`Scanner` 必须包含如下函数：

- `__iter__`: 用于返回迭代器本身
- `next`: 返回迭代器中下一个元素，否则抛出停止迭代异常

剩下的函数则不是框架强制要求的，`open` 函数用于打开一个源文件、`check_type` 函数用于检查 `token` 是什么类型。词法分析器遵循的规则参看`PL0.pdf` 或附录中 3.2 节 -Complete List of Reserved Words and Tokens。需要注意的是，所有关键字都需要全部大写。`Token` 必须拥有 `self.token_type`，否则 LR 分析器无法确定 `token` 具体是什么。

2.3 LR 主控程序

LR 主控程序文件名为`lrparser.py`，这个程序是整个编译器框架的核心，接受 `xml` 文件、语法分析器、词法分析器作为配置。这种情况中，`xml` 文件就是 LR 分析器的分析表。分析表遵循的语法规则见`PL0.pdf` 或附录中 3.1 节 -PL/0 EBNF Grammar。本文件中定

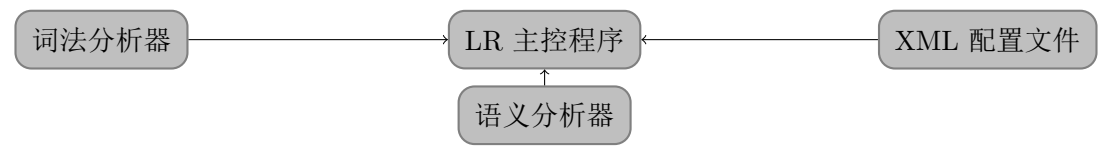


图 2: 配置文件

义了两种异常，`XmlDecodeError` 与 `ParseringError`，对应着 XML 解析错误与语法分析错误。`Transition`、`Rule` 两个类定义了 LR 分析器必要的数据结构。`Parser` 类定义了 LR 分析器，函数功能如下：

函数	参数	功能
<code>open</code>	<code>name:string</code>	用于打开名为 <code>name</code> 的 XML 配置文件，将文件中的 LR 分析表载入内存，供分析使用
<code>state_transit</code>	<code>token:Token</code>	接受当前的 <code>Token</code> ，用于在不同的状态中进行转换的函数
<code>parser</code>	<code>scanner:Scanner</code> , <code>token_class:Class</code> , <code>syntaxer:Syntaxer</code>	接受 <code>scanner</code> 为词法分析器的一个实例， <code>token_class</code> 为 <code>Token</code> 类， <code>syntaxer:Syntaxer</code> 为语义分析器的一个实例，并进行语法分析的函数

表 2: LR 分析器中函数用途

2.4 语义分析器

本文件为 `syntaxer.py`，其中类较多，原因在于每个非终结符都需要一个类，这样可以更好的规范框架。本程序中最主要的类为 `Syntaxer`，这个类定义了语义分析器。在框架中，语义分析器必须拥有如下函数：`token_to_terminal`：接受 `token` 转换为终结符、`get_reduce_process_function`：接受产生式的编号返回相应的产生式处理函数，不同产生式的编号请查看 Bison 的生成文件、`error_handler`：传入出错的 `token` 与 `state`，显示错误。对于每个产生式处理函数，都接受一个元组类型变量 `rhs`，为产生式的右端，`rhs` 从 0 开始，依次向右编号，为每个产生式右部的项；处理函数必须有一个返回，为产生式左侧元素。由于 LR 分析器拥有较多状态，所以出错信息相对较简单。本分析器设计思想为语法制导，使用 S-属性文法，仅使用综合属性。

2.5 语义分析后端

本文件为 `backend.py`，维护了符号表等编译时需要的数据结构，同时也参与中间代码 P-Code 的生成。使用的中间代码为 PL/0 的 P-Code，定义见附带的 `PL0.pdf` 或附录中 3.4 节 -PL/0 Instruction Set Architecture。这种中间代码假定计算机有一个无穷大的栈作为唯一存储空间。在这个表格中，去除了 SIO 指令，添加了 OPR 0,14 指令作为输出栈顶元素的指令，OPR 0,15 指令作为输出回车的指令，OPR 0,16 指令为输入数字，并压栈。

2.6 中间代码翻译器

文件为 `translate.py`。可以将 P-Code 翻译成针对 8086 的 Intel 汇编风格代码。

3 规范性

由于本框架接近简单的 `yacc`，对各类文件有规范性要求。

3.1 XML 文件规范性

XML 文件遵循[本链接¹](#)的规范。XML 的根节点可以是其他名称，也可以拥有其他属性，用来标识不同的 XML 生成器。但是相应的子节点必须遵守 Bison 兼容的 XML 语法要求，否则无法识别。通过 Bison 生成 XML 文件的方法是：`bison -x $file_name$`。

3.2 XML 文件与语法分析器配合的规范性

由于 Bison 会对每个产生式编号，这个编号是实现语义分析器重要的一部分，语义分析器中的 `get_reduce_process_function` 函数会接收这个编号来确定返回那个规则的处理函数。所以需要查看 Bison 的 output 信息，使用如下命令生成这个文件：`bison -v $file_name$`。

¹<http://wojciechpolak.org/notes/bison-xml-report.html>

4 使用方法

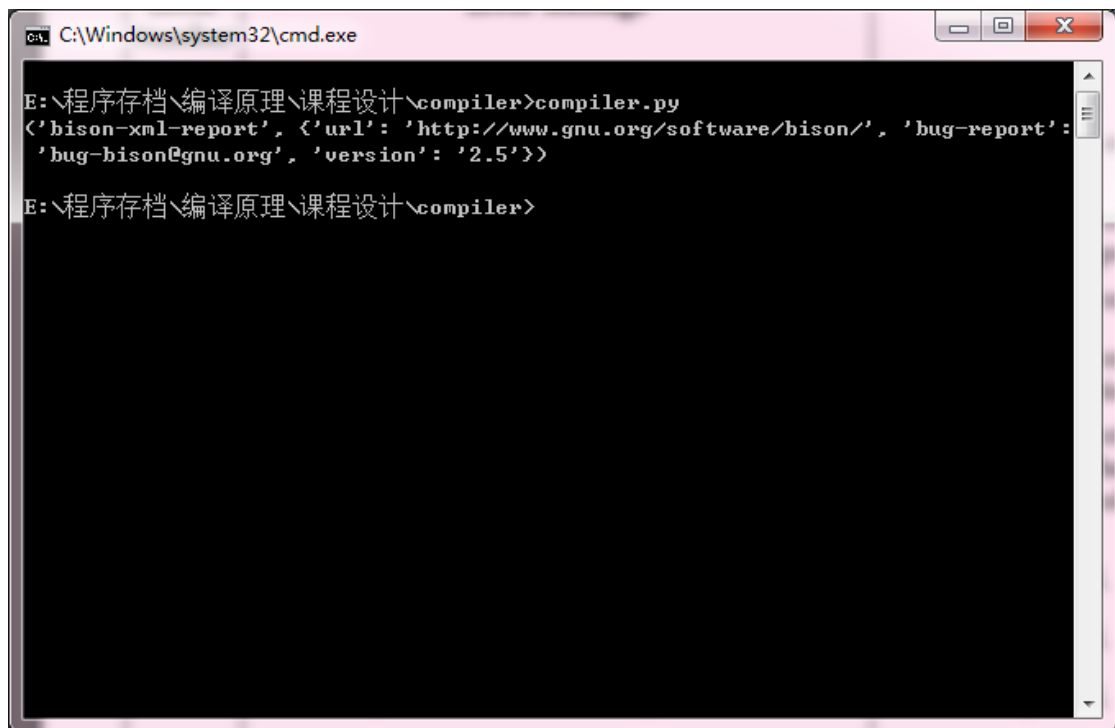
本程序在编译前端支持 READ 语句 (中间代码为 OPR 0, 16), 但是在编译后端不支持。执行 `compiler.py $name$` 从 `name` 中读取 PL/0 代码, 若没有参数, 默认为 `test.pas`, 正常情况下程序只会输出 XML 文件的根节点信息。程序将中间代码输出到 `out.txt` 中, 将目标代码生成到 `res.asm` 中。生成的中间代码为 P-Code, 可以被 Simulator 目录下的 `main.exe` 解释执行。`main.exe` 使用 `in.txt` 作为中间代码的输入, `out.txt` 作为中间代码的执行结果输出。注意 `in.txt` 需要在文件第一行执行代码行数, 并将需要输入的数字按顺序放到代码之后。

5 出错处理

如果代码中有不符合语法、语义的部分, 编译器将会输出错误所在位置。如果错误涉及重复定义、调用未定义函数、变量等, 会输出详细错误信息。例如在第三行第五列出错, 会输出: `Error on 3:5, State:?`, 表明出错位置与出错时 LR 分析器所在的状态。

6 运行样例

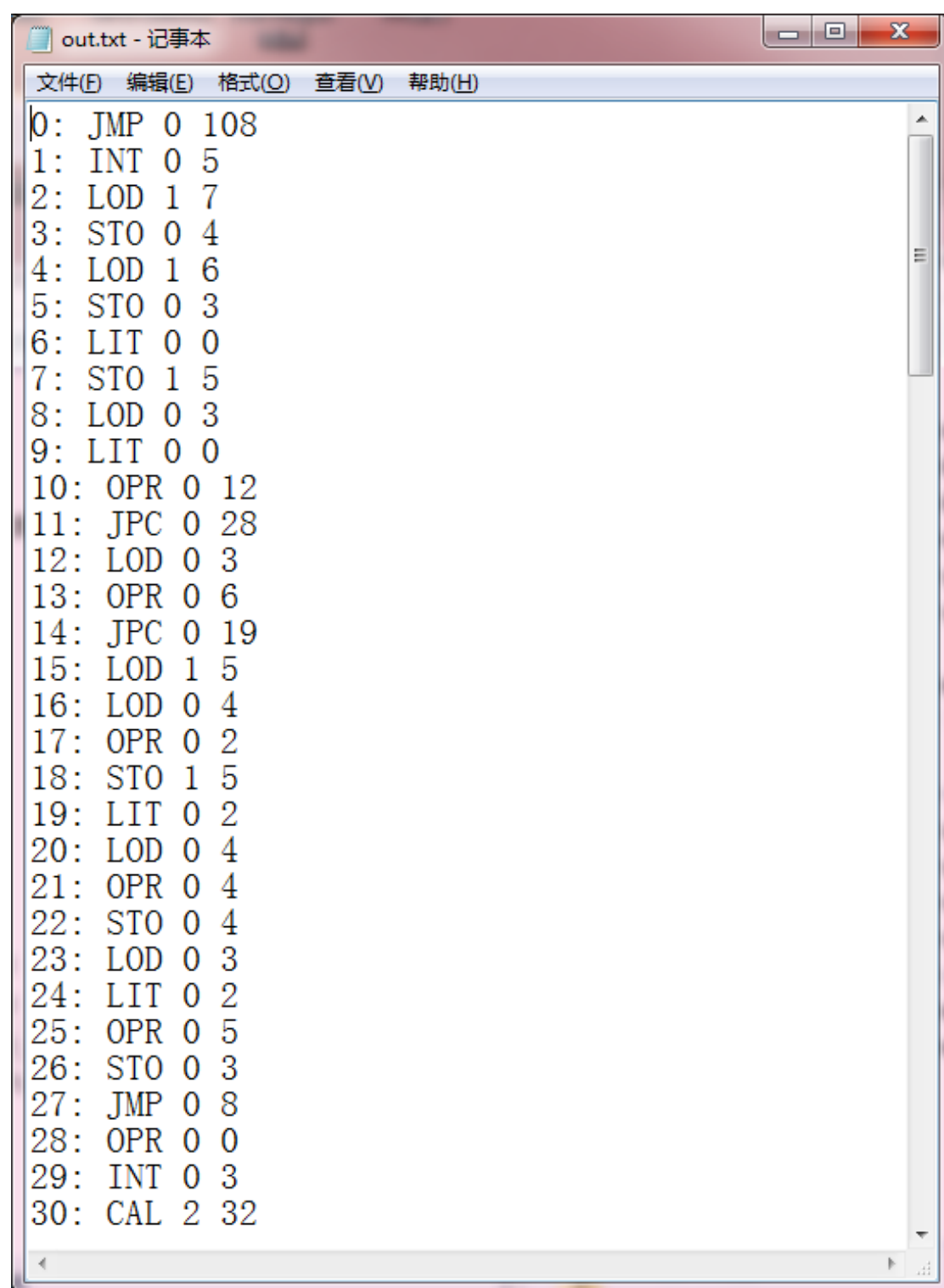
首先安装 CPython 环境, 不做演示。然后运行 `compiler.py`, 得出目标代码 `res.asm` 与中间代码 `out.txt`, 然后将 `res.asm` 拷贝到 `emu8086` 中运行, 结果如下:



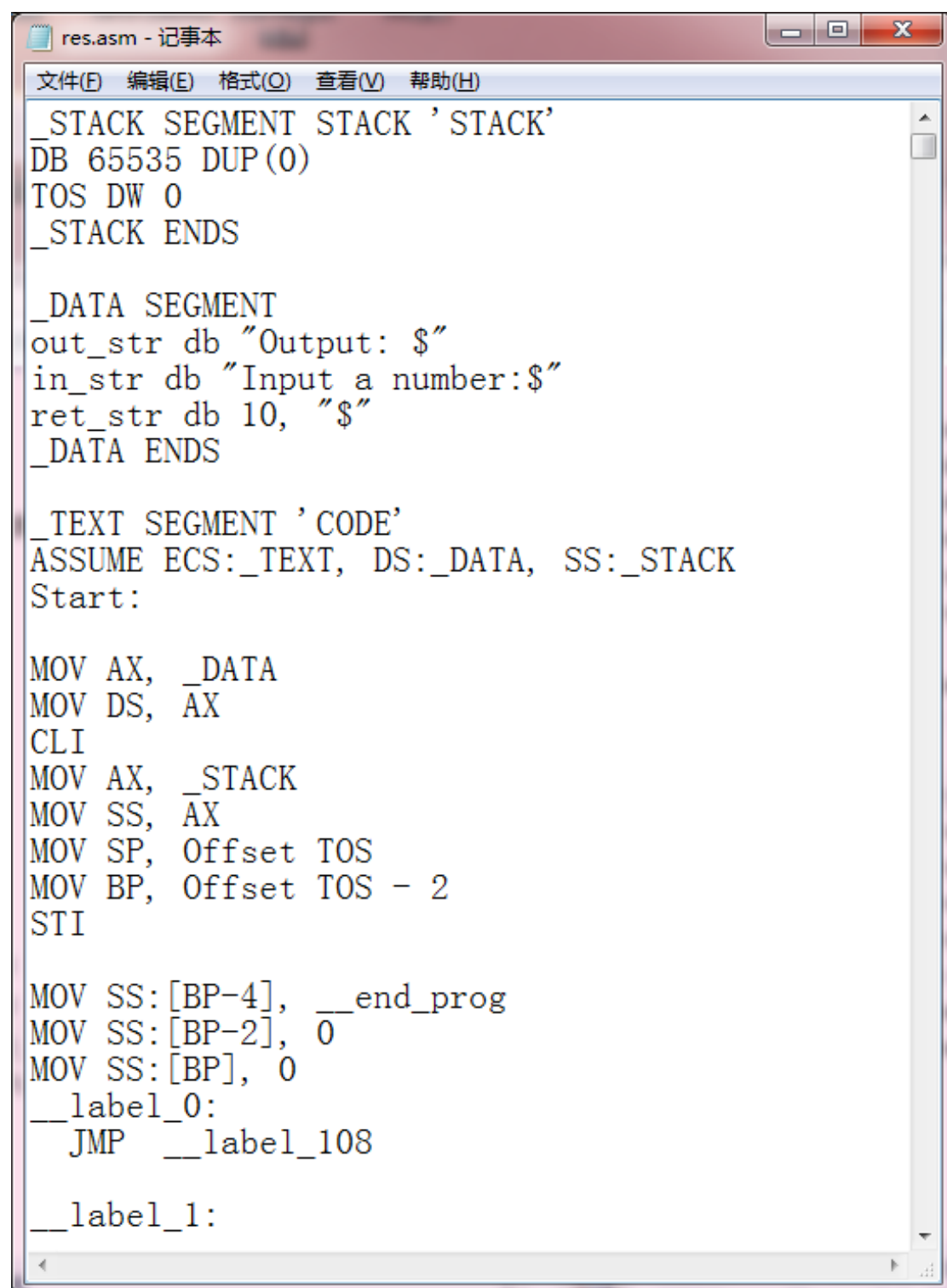
```
C:\Windows\system32\cmd.exe

E:\程序存档\编译原理\课程设计\compiler>compiler.py
(<'bison-xml-report', {'url': 'http://www.gnu.org/software/bison/', 'bug-report':
'bug-bison@gnu.org', 'version': '2.5'})

E:\程序存档\编译原理\课程设计\compiler>
```



```
out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
0: JMP 0 108
1: INT 0 5
2: LOD 1 7
3: STO 0 4
4: LOD 1 6
5: STO 0 3
6: LIT 0 0
7: STO 1 5
8: LOD 0 3
9: LIT 0 0
10: OPR 0 12
11: JPC 0 28
12: LOD 0 3
13: OPR 0 6
14: JPC 0 19
15: LOD 1 5
16: LOD 0 4
17: OPR 0 2
18: STO 1 5
19: LIT 0 2
20: LOD 0 4
21: OPR 0 4
22: STO 0 4
23: LOD 0 3
24: LIT 0 2
25: OPR 0 5
26: STO 0 3
27: JMP 0 8
28: OPR 0 0
29: INT 0 3
30: CAL 2 32
```



```
res.asm - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

_STACK SEGMENT STACK 'STACK'
DB 65535 DUP(0)
TOS DW 0
_STACK ENDS

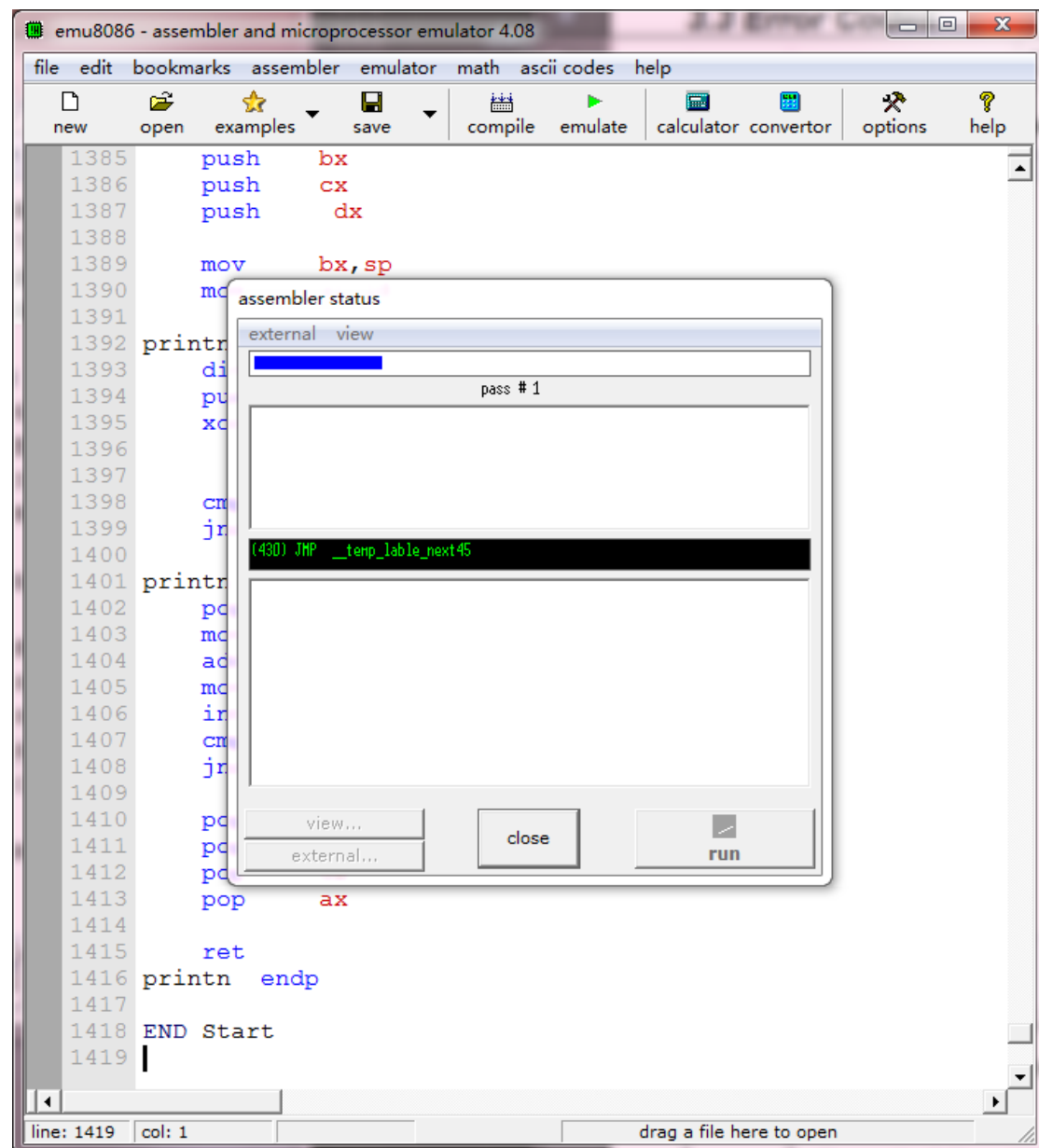
_DATA SEGMENT
out_str db "Output: $"
in_str db "Input a number: $"
ret_str db 10, "$"
_DATA ENDS

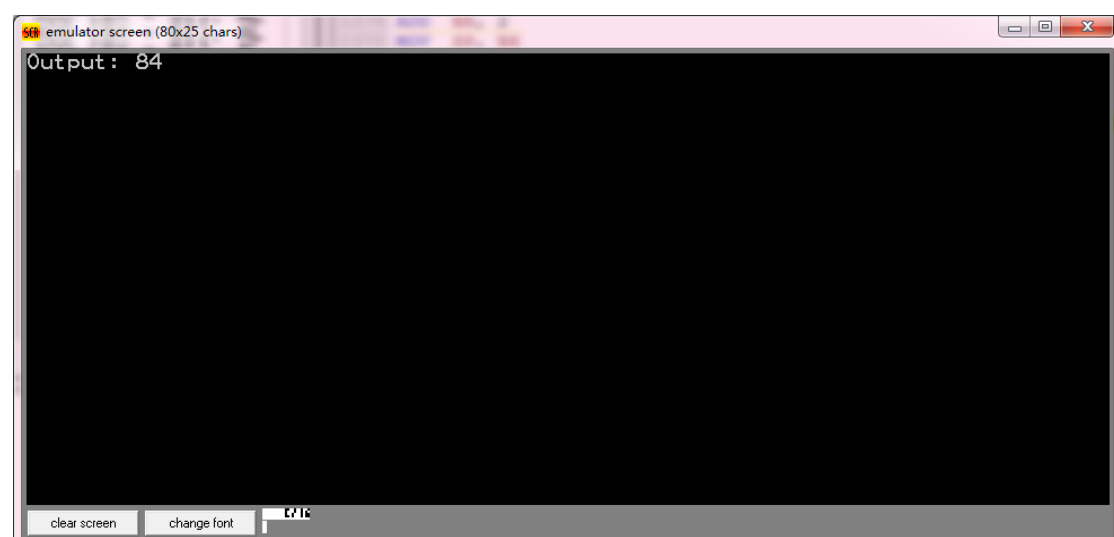
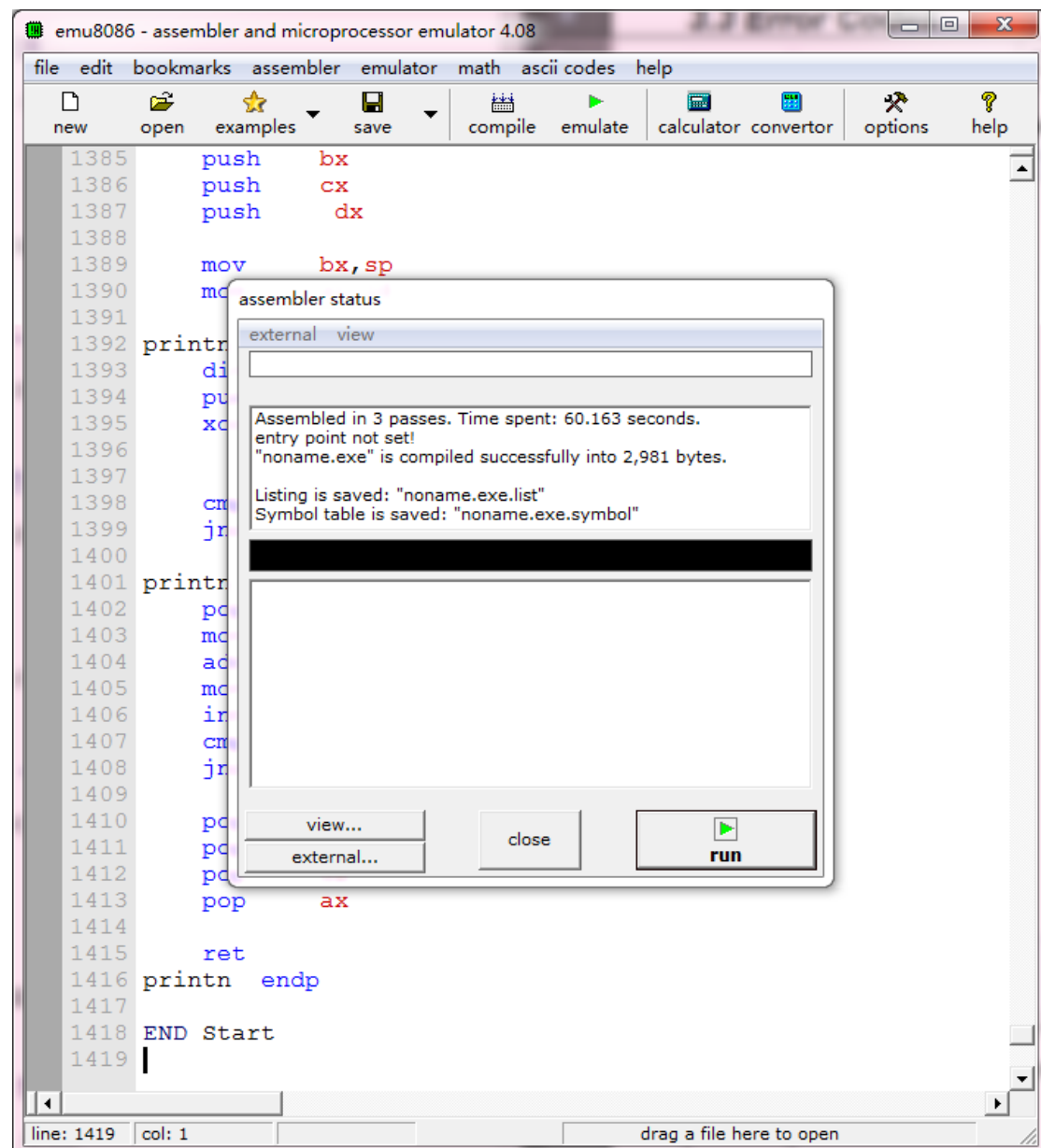
_TEXT SEGMENT 'CODE'
ASSUME ECS:_TEXT, DS:_DATA, SS:_STACK
Start:

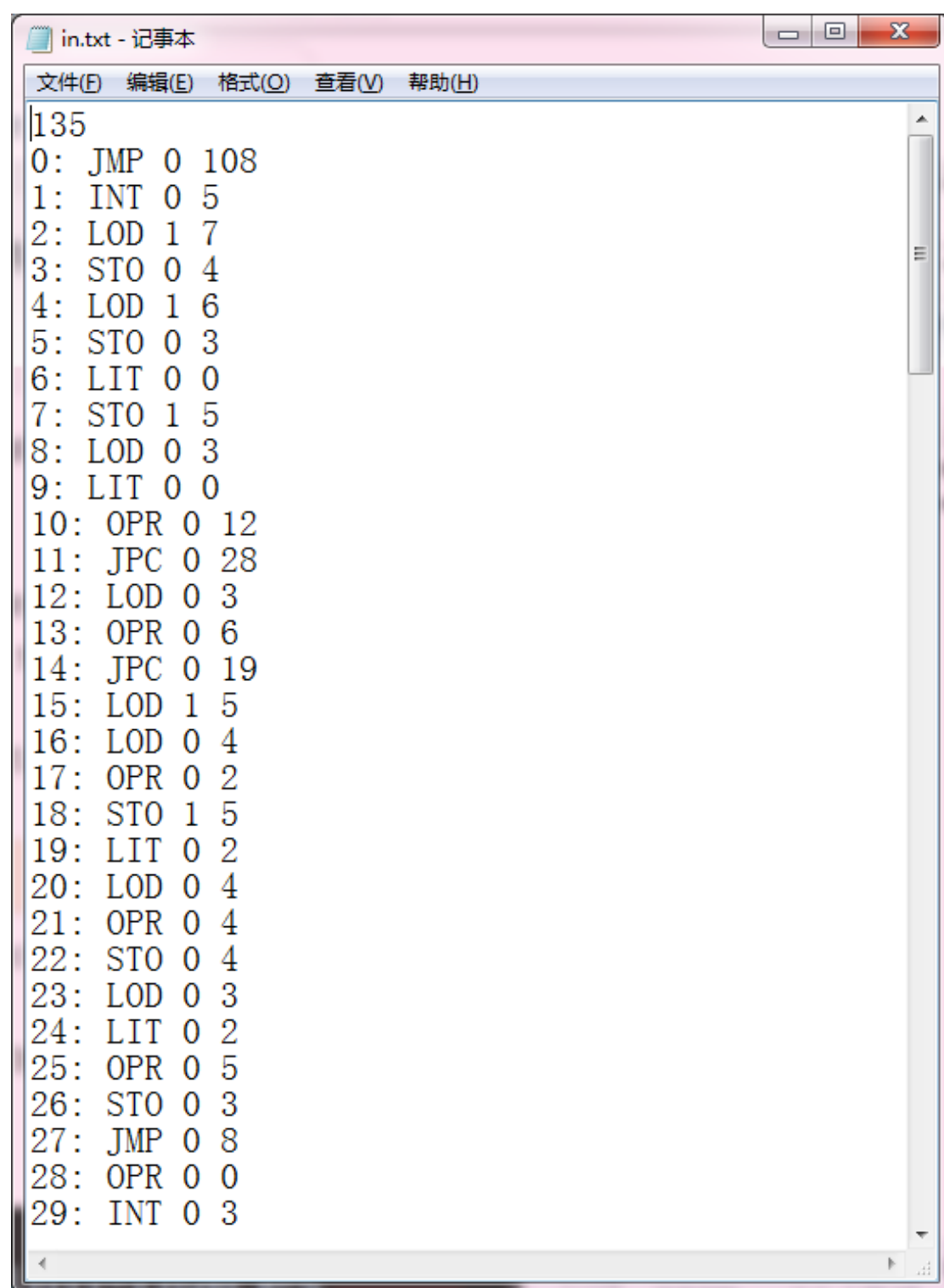
MOV AX, _DATA
MOV DS, AX
CLI
MOV AX, _STACK
MOV SS, AX
MOV SP, Offset TOS
MOV BP, Offset TOS - 2
STI

MOV SS:[BP-4], __end_prog
MOV SS:[BP-2], 0
MOV SS:[BP], 0
__label_0:
    JMP __label_108

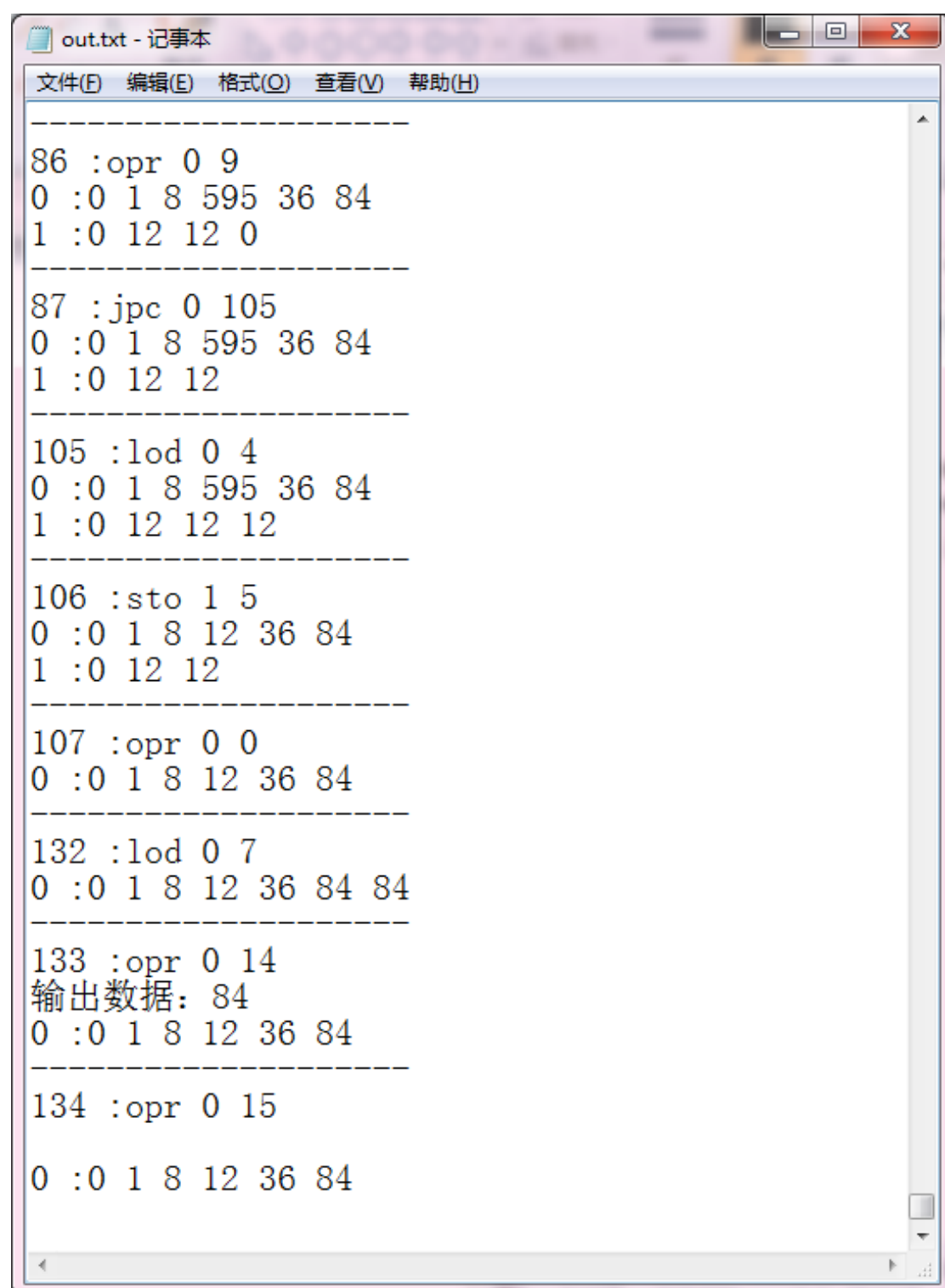
__label_1:
```







```
in.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
|135
0: JMP 0 108
1: INT 0 5
2: LOD 1 7
3: STO 0 4
4: LOD 1 6
5: STO 0 3
6: LIT 0 0
7: STO 1 5
8: LOD 0 3
9: LIT 0 0
10: OPR 0 12
11: JPC 0 28
12: LOD 0 3
13: OPR 0 6
14: JPC 0 19
15: LOD 1 5
16: LOD 0 4
17: OPR 0 2
18: STO 1 5
19: LIT 0 2
20: LOD 0 4
21: OPR 0 4
22: STO 0 4
23: LOD 0 3
24: LIT 0 2
25: OPR 0 5
26: STO 0 3
27: JMP 0 8
28: OPR 0 0
29: INT 0 3
```



```
out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

-----
86 :opr 0 9
0 :0 1 8 595 36 84
1 :0 12 12 0
-----
87 :jpc 0 105
0 :0 1 8 595 36 84
1 :0 12 12
-----
105 :lod 0 4
0 :0 1 8 595 36 84
1 :0 12 12 12
-----
106 :sto 1 5
0 :0 1 8 12 36 84
1 :0 12 12
-----
107 :opr 0 0
0 :0 1 8 12 36 84
-----
132 :lod 0 7
0 :0 1 8 12 36 84 84
-----
133 :opr 0 14
输出数据: 84
0 :0 1 8 12 36 84
-----
134 :opr 0 15
0 :0 1 8 12 36 84
```

A 附录 –设计手册

PL/0 User's Manual

Author: Adam Dunson
Last updated: 14 April 2012

1.0 Programming in PL/0

PL/0 is a fairly simple language that supports constants, integers, and procedures. PL/0 programs have the following structure:

1. constant definitions
2. variable declarations
3. procedure declarations
 - a. subroutine definition, same as program structure
4. statement(s)

For the most part, whitespace is ignored (except in certain cases in order to differentiate between reserved keywords and identifiers). Additionally, anything between comment delimiters, e.g., `/*` and `*/`, will be ignored. Finally, PL/0 programs must end with a period.

Figure 1 shows an example of a simple PL/0 program with comments:

```

int foo;
begin
  foo := /*here is a comment*/ 1;

  /*comments
  can
  span
  multiple
  lines*/

  out foo;
end.

```

Figure 1: Program with comments

1.1 Datatypes

Currently, this implementation of PL/0 supports the following datatypes:

- constants (`const`)
- integers (`int`)
- procedures (`procedure`)

An identifier is used to refer to specific instances of each datatype. Identifiers must be no more than 11 characters in length, must begin with a character, may contain uppercase and lowercase letters as well as numbers, and must not be any of the reserved keywords listed in Appendix A.

In addition to identifiers, number literals are used through the program for arithmetic and other operations. Number literals must be integers and must be no more than 5 digits long. We do not currently support negative number literals.

1.1.1 Constants

Constants are integer types. They may only be defined once per program. Constants are immutable; that is, you may not assign values to them after they have been defined. You may define more than one constant at a time by separating the identifiers by commas. Constant definitions must end with a semicolon.

Constants are defined using the following syntax:

```
const F00 = 1, BAR = 2;
```

After you define a constant, you may use them throughout your program as you would an ordinary number. The compiler converts constants to their respective values upon compilation, so the following example,

```
foo := F00 + BAR;
```

is equivalent to this:

```
foo := 1 + 2;
```

1.1.2 Integers

Integers are mutable; meaning, you can define and reassign their values later on. Integers are declared at the top of the file after any constants (if there are any), and you may declare more than one at a time. Integers are defined with values as constants are. Instead, you must assign integers values using the assignment operator `:=` (as seen above). Here is an example of how to declare integers in PL/0:

```
int foo, bar, baz;
```

You may assign variables values of constants or other variables, number literals, or expressions (see section 1.2.2).

1.1.3 Procedures

Procedures can be thought of as subroutines or sub-programs. They contain nearly everything that a program could contain. **Please be aware that this version of PL/0 currently supports up to 10 levels of nested procedures (1 Main + 10 Procedures).** Figure 2 shows a procedure that prints the numbers 1 through 10:

```

procedure count10;
  int a;
  begin
    a := 1;
    while a <= 10 do
      begin
        out a;
        a := a + 1;
      end;
    end;
  end;
```

Figure 2: Procedure that counts 1 to 10

1.2 Expressions

Expressions get their name from mathematical expressions which represent or return a value. Expressions can be composed of constant or variable identifiers, number literals, or the arithmetic symbols `+`, `-`, `*`, `/`, `(`, and `)`. PL/0 follows the standard order of operations when calculating the value of an expression.

1.3 Statements

Statements are how the program gets things done. Except for the last statement in a block, statements must end with a semicolon.

1.3.1 Input/Output

Input is handled by using the `in` keyword followed by a variable identifier (you cannot use a constant or procedure identifier), e.g.,

```
in foo;
```

This will assign whatever value the user inputs to `foo`.

Output is handled by using the `out` keyword followed by a constant or variable identifier, a number literal, or an expression. Here are a few examples:

```
out foo;
out 42;
out (1 + 2) * (3 + 4);
```

1.3.2 Blocks

Blocks are collections of statements, each of which are separated by a semicolon. Blocks are denoted by the `begin` and `end` keywords. See section 1.1.3 (Procedures) for how a block can be nested inside of other statements.

1.3.3 Assignment

As mentioned before, variables can be assigned values by using constant or variable identifiers, number literals, or expressions. The assignment operator, `:=`, only works for variables inside of a statement. Currently, you are not able to assign variables initial values at the time of declaration.

1.3.4 Conditionals

To conditionally execute code, use the conditional keywords `if`, `then`, and `else`. These allow you to check a condition and, if true, execute some code, or else execute some other code. A conditional expression can either be two expressions separated by a relational operator (e.g., expression `[rel-op]` expression) or else using the unary `odd` keyword:

```
odd x + y
```

The `odd` keyword will return true if the expression evaluates to an odd number, or else it will return false if the expression evaluates to an even number.

Valid relational operations are as follows:

- = (equal)
- <> (not equal)
- < (less than)
- <= (less than or equal)
- > (greater than)
- >= (greater than or equal to)

Here is an example of an if-then without an else:

```
if 1 = 1 then out 1;
```

And with an else:

```
if 1 <> 1 then out 1 else out 0;
```

Conditionals can be nested in one another. One such use is checking multiple conditions before executing code. For example, here's a snippet from a basic four-function calculator program:

```
if op = ADD then call add
else if op = SUB then call sub
else if op = MULT then call mult
else if op = DIV then call div
else done := 1;
```

1.3.5 Loops

Loops are another useful construct delimited by the `while` and `do` keywords. Often you will need to iterate over a range of numbers, or perhaps to perform the same set of instructions until some condition is false. From the same four-function calculator program used above, Figure 3 shows an example of a while loop.

```
while done = 0 do
begin
  in op;
  in y;
  if op = ADD then call add
  else if op = SUB then call sub
  else if op = MULT then call mult
  else if op = DIV then call div
  else done := 1;
end;
```

Figure 3: while loop example

1.3.6 Calling Procedures

To invoke a procedure, use the `call` keyword. There is no way to explicitly pass arguments to a procedure. However, procedures have access to any variables and procedures that are declared within their scope.

For instance, Figure 4 (right) shows the complete source code for the calculator program. Notice that the procedures are able to use the `x`, `y`, and `done` variables because they were declared within their scope.

Using the Basic Calculator Program

First, the program will ask the user to input the first value, `x`.

Next, the program will ask the user to input an operation, `op`. This can be any one of 1, 2, 3, or 4 (add, subtract, multiply, or divide, respectively). Inputting anything else will cause the program to exit.

Then, the program will ask the user to input the second value, `y`.

Finally, the program will call the procedure corresponding to `op`. This will assign a new value to `x` and will also output this value.

The program returns to asking the user to input an operation. It will continue this process until the user inputs any value other than 1, 2, 3, or 4 for the `op`.

```
const ADD = 1, SUB = 2, MULT = 3, DIV = 4;
int op, x, y, done;

procedure add;
begin
  x := x + y;
  out x;
end;

procedure sub;
begin
  x := x - y;
  out x;
end;

procedure mult;
begin
  x := x * y;
  out x;
end;

procedure div;
begin
  /* check for divide-by-zero errors */
  if y <> 0 then
    begin
      x := x / y;
      out x;
    end
  else done := 1;
end;

begin
  done := 0;
  in x;

  while done = 0 do
    begin
      in op;
      if op < 1 then done := 1
      else if op > 4 then done := 1;

      if done = 0 then
        begin
          in y;

          if op = ADD then call add
          else if op = SUB then call sub
          else if op = MULT then call mult
          else if op = DIV then call div
          else done := 1;
        end;
      end;
    end;
  end.
```

Figure 4: Complete source code for basic four-function calculator program in PL/0.

1.4 Advanced Examples

PL/0 supports recursive and nested procedures. Nested procedures introduce some nuances into the concept of scope that might be less than obvious.

1.4.1 Recursive Procedures

Figure 5 shows a program that uses a recursive procedure to calculate the factorial of a user-input integer.

```

int f, n;
procedure fact;
  int ans1;
  begin
    ans1:=n;
    n:= n-1;
    if n < 0 then f := -1
    else if n = 0 then f := 1
    else call fact;
    f:=f*ans1;
  end;

begin
  in n;
  call fact;
  out f;
end.

```

Figure 5: Recursive procedure example

1.4.2 Nested Procedures

Nested procedures give the programmer more control over where procedures may be called from.

Figure 6 on the next page is a modified version of the basic four-function calculator program example from before. Notice that the main block now calls `calculate`, which in turn uses `op` to determine which of the nested procedures to call. Be aware that you cannot call `add`, `sub`, `mult`, or `div` from inside the main block as you could before. Finally, notice that `mult` now calls `add` (this is to demonstrate scope). The variable `C` cannot be accessed outside of `mult` due to scope.

```

const ADD = 1, SUB = 2, MULT = 3, DIV = 4;
int op, x, y, done;

procedure calculate;
  procedure add;
    begin
      x := x + y;
    end;
  procedure sub;
    begin
      x := x - y;
    end;
  procedure mult;
    int c;
    begin
      c := y - 1;
      y := x; /* resets y argument for calling add */
      while c > 0 do
        begin
          call add;
          c := c - 1;
        end;
      /* old method: x := x * y; */
    end;
  procedure div;
    begin
      /* check for divide-by-zero errors */
      if y <> 0 then
        begin
          x := x / y;
        end
      else done := 1;
    end;
  begin
    if op = ADD then call add
    else if op = SUB then call sub
    else if op = MULT then call mult
    else if op = DIV then call div
    else done := 1;
    if done = 0 then out x;
  end;

begin
  done := 0;
  in x;
  while done = 0 do
    begin
      in op;
      if op < 1 then done := 1
      else if op > 4 then done := 1;

      if done = 0 then
        begin
          in y;
          call calculate;
        end;
    end;
  end;
end.

```

*Figure 6: Modified
calculator
program using
nested procedures*

2.0 Compiling and Executing Programs Written in PL/0

The `pl0-compiler` program is both a compiler and a virtual machine for PM/0 (the machine for which the PL/0 ISA was designed).

2.1 Building *pl0-compiler*

These instructions assume you have experience using a terminal. You will need GCC and GNU Make prior to building `pl0-compiler`.

To build the compiler's executable file, do the following:

1. Obtain a copy of the source code for `pl0-compiler`
2. Open a terminal (or command prompt) and navigate to the project directory
 - You should see a file called `README.text` and another called `Makefile`.
3. Run `make`
 - This will output a file called `pl0-compiler` (or `pl0-compiler.exe`) into the `bin/` directory.

Once you have an executable `pl0-compiler`, you are ready to run PL/0 programs.

2.2 Running PL/0 Programs with *pl0-compiler*

To begin, open a terminal (or command prompt) and navigate to wherever your `pl0-compiler` is located. The default mode is to display only `in/out` calls from the PL/0 program. To do this, run the following:

```
./pl0-compiler /path/to/your/file
```

If you want to see more output, there are three command-line flags available that you can use.

- The `-l` flag instructs `pl0-compiler` to display the internal representation of the PL/0 program. That is, it displays the token file including both a raw and a pretty format.
- The `-a` flag instructs `pl0-compiler` to display the generated assembly code in both a raw and pretty format.
- The `-v` flag instructs `pl0-compiler` to display a stack trace while the virtual machine executes your program.

These flags may be used in any combination for more or less output. To use more than one flag, you can run something like this:

```
./pl0-compiler -l -a -v /path/to/your/file
```

or else, like this:

```
./pl0-compiler -lav /path/to/your/file
```

The only restriction is that the filename of your PL/0 program must be the last argument.

3.0 Reference

3.1 PL/0 EBNF Grammar

```

program ::= block "." .
block ::= const-declaration var-declaration procedure-declaration statement
.
const-declaration ::= ["const" ident "=" number {""," ident "=" number} ";"]
.
var-declaration ::= [ "int" ident {""," ident} ";"] .
procedure-declaration ::= { "procedure" ident ";" block ";" }
statement ::= [ ident ":" expression
| "call" ident
| "begin" statement { ";" statement } "end"
| "if" condition "then" statement ["else" statement]
| "while" condition "do" statement
| "read" ident
| "write" expression
| e ] .
condition ::= "odd" expression
| expression rel-op expression .
rel-op ::= "=" | "<>" | "<" | "<=" | ">" | ">=" .
expression ::= [ "+" | "-" ] term { ("+" | "-") term } .
term ::= factor { ("*" | "/" ) factor } .
factor ::= ident | number | "(" expression ")" .
number ::= digit {digit} .
ident ::= letter {letter | digit} .
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
letter ::= "a" | "b" | ... | "y" | "z" | "A" | "B" | ... | "Y" | "Z" .

```

Figure 7: PL/0 EBNF Grammar

3.2 Complete List of Reserved Words and Tokens

Symbol	Internal Name	Internal Value	Usage
	nulsym	1	reserved
	identsym	2	constant, variable, and procedure identifiers
	numbersym	3	number literals
+	plussym	4	addition in expressions
-	minussym	5	subtraction in expressions
*	multsym	6	multiplication in expressions
/	slashsym	7	division in expressions
odd	oddsym	8	determining if an expression is odd
=	eqlsym	9	constant definitions, checking the equality of two expressions

<>	neqsym	10	checking that two expressions are not equal
<	lessym	11	checking that the left expression is less than the right expression
<=	leqsym	12	checking that the left expression is less than or equal to the right expression
>	gtrsym	13	checking that the left expression is greater than the right expression
>=	geqsym	14	checking that the left expression is greater than or equal to the right expression
(lparentsym	15	begin a factor
)	rparentsym	16	end a factor
,	commasym	17	separate constant, variable identifiers in their respective declarations
;	semicolonsym	18	end statements
.	periodsym	19	end of program
:=	becomesym	20	variable assignments
begin	beginsym	21	begin a block of statements
end	endsym	22	end a block of statements
if	ifsym	23	begin an if-then statement, followed by a condition
then	thensym	24	part of if-then, followed by a statement
while	whilesym	25	begin while loop, followed by a condition
do	dosym	26	part of while loop, followed by a statement
call	callsym	27	calls a procedure
const	constsym	28	begin constant declarations
int	intsym	29	begin integer declarations
procedure	procsym	30	begin a procedure declaration
out	outsym	31	output the value of an expression
in	insym	32	ask the user to input a value and assign it to a variable
else	elsesym	33	optionally follows if-then statements

Table 1: Complete List of Reserved Words and Tokens

3.3 Error Codes

Error Number	Error Message	Comments/Suggestions
0	No errors, program is syntactically correct.	N/A
1	Use = instead of :=.	You tried to assign a value to a variable using =.
2	= must be followed by a number.	Syntax error near constant declarations or in a conditional expression.
3	Identifier must be followed by =.	Syntax error near constant declarations.
4	const, int, procedure must be followed by identifier.	Syntax error near constant, variable, or procedure declarations.
5	Semicolon or comma missing.	You missed a semicolon or a comma somewhere. Also check that you aren't adding extra semicolons to if-then-else and while-do's.
6	Incorrect symbol after procedure declaration.	Not currently used.
7	Statement expected.	Not currently used.
8	Incorrect symbol after statement part in block.	Not currently used.
9	Period expected.	Missed a period at the end of the program.
10	Semicolon between statements missing.	Except for the last one in a block, every statement needs to end with a semicolon.
11	Undeclared identifier.	You tried to use an undeclared constant, variable, or procedure, or you tried to access something that is outside of your scope.
12	Assignment to constant or procedure is not allowed.	You tried to assign a value to a constant or a procedure. Check your variable names.
13	Assignment operator expected.	You began a statement with an identifier, but it wasn't followed by an assignment operator (:=).
14	call must be followed by an identifier.	You used call, but you didn't include the procedure name.
15	Call of a constant or variable is meaningless.	You tried to call a constant or a variable, which is meaningless.
16	then expected.	if [condition] must be followed by then [statement].
17	Semicolon or end expected.	Not currently used.

18	do expected.	while [condition] must be followed by do [statement].
19	Incorrect symbol following statement.	Not currently used.
20	Relational operator expected.	In a conditional expression, you are missing a relational operator.
21	Expression must not contain a procedure identifier.	You cannot use procedures in expressions (since they do not return or represent values).
22	Right parenthesis missing.	Missing the right parenthesis at the end of a factor.
23	The preceding factor cannot begin with this symbol.	There is something wrong with a factor used in an expression.
24	An expression cannot begin with this symbol.	Not currently used.
25	This number is too large.	Code generator exceeded the maximum number of lines of code.
26	out must be followed by an expression.	You used out, but didn't specify anything to output.
27	in must be followed by an identifier.	You used in, but you didn't specify what variable to assign it to.
28	Cannot reuse this symbol here.	Not currently used.
29	Cannot redefine constants.	Constants cannot be redefined.

Table 2: Error codes

3.4 PL/0 Instruction Set Architecture

All PL/0 instructions are of the form **OP L, M** where **OP** is the op code, **L** is the lexicographical level, and **M** is an address, data, or an ALU operation.

Op Code	Syntax	Description
1	LIT 0, M	Push constant value (literal) M onto the stack
2	OPR 0, M	Operation to be performed on the data at the top of the stack
	OPR 0, 0	Return ; used to return to the caller from a procedure.
	OPR 0, 1	Negation ; pop the stack and return the negative of the value
	OPR 0, 2	Addition ; pop two values from the stack, add and push the sum
	OPR 0, 3	Subtraction ; pop two values from the stack, subtract second from first and push the difference
	OPR 0, 4	Multiplication ; pop two values from the stack, multiply and push the product
	OPR 0, 5	Division ; pop two values from the stack, divide second by first and push the quotient

	OPR 0, 6	Is odd? (divisible by two); pop the stack and push 1 if odd, 0 if even
	OPR 0, 7	Modulus ; pop two values from the stack, divide second by first and push the remainder
	OPR 0, 8	Equality ; pop two values from the stack and push 1 if equal, 0 if not
	OPR 0, 9	Inequality ; pop two values from the stack and push 0 if equal, 1 if not
	OPR 0, 10	Less than ; pop two values from the stack and push 1 if first is less than second, 0 if not
	OPR 0, 11	Less than or equal to ; pop two values from the stack and push 1 if first is less than or equal second, 0 if not
	OPR 0, 12	Greater than ; pop two values from the stack and push 1 if first is greater than second, 0 if not
	OPR 0, 13	Greater than or equal to ; pop two values from the stack and push 1 if first is greater than or equal second, 0 if not
3	LOD L, M	Load value to top of stack from the stack location at offset M from L lexicographical levels down
4	STO L, M	Store value at top of stack in the stack location at offset M from L lexicographical levels down
5	CAL L, M	Call procedure at code index M
6	INC 0, M	Increment the stack pointer by M (allocate M locals); by convention, this is used as the first instruction of a procedure and will allocate space for the Static Link (SL) , Dynamic Link (DL) , and Return Address (RA) of an activation record
7	JMP 0, M	Jump to instruction M
8	JPC 0, M	Pop the top of the stack and jump to instruction M if it is equal to zero
9	SIO 0, 1	Start I/O ; pop the top of the stack and output the value
10	SIO 0, 2	Start I/O ; read input and push it onto the stack

Table 3: PL/0 Instruction Set Architecture