StackBOOM
语言设计说明书

程序设计语言原理大作业

苏星熠（ZY2206148） 黄浩程（ZY2206134）

2022年12月

1. 设计背景和目标

1.1 设计背景

经过一学期的学习，我们学习得到了许许多多不同设计思想的语言，包括过程式、对象式、函数式、逻辑式等等。事实上，在使用了多年面向对象的语言以后，再度上手其他思想的语言让我们感到十分“不快”——或许从语言排行的名单来看也能够发现问题，毕竟使用最多的语言，几乎都是面向对象的语言。

而面对这些发展已然成熟、各个几乎都比我们年龄还大的语言，我们实际上很难提出哪些对于已有语言的改进，如果非要提出来一个，那就是我非常想给Python语言的语句结尾增加分号。所以，在大作业的逼迫下，我们思考了一个问题：是否存在其他的语言设计思想。

实际上思考一下，程序语言无非有两个部分：数据和操作。对象式语言把这两个打了个包叫对象，所以能够万物皆对象；函数式语言把操作看得比数据重要，所以“函数成为了一等公民”；过程式语言相当于数据和操作平权。那么，存不存在一种语言，将数据的权重进行放大呢？

然后我们想到了计算器上的堆栈式表示法，或者说逆波兰表示法。在计算时，对于数据会存入栈中，先入后出，对于操作会直接进行计算，用后即弃，在某种意义上，也算是提高了数据的地位。

查阅相关资料后，本来欣喜若狂以为自己进行了创新，结果发现，果然是读书太少了。上述设计思想指导的语言已经经历了发展、鼎盛再到衰落的过程。典型的语言有RPL、PostScript语言。前者应用于惠普公司制造的计算器产品，后者则由Adoby公司制作，首先应用于打印机产品，后用于文档页面的描述，并对后续PDF的诞生有很大影响。

上述两个语言有一些共同：专用性很强，并且大部分情况下，语言的代码经由计算机自己生成。这导致语言本身还是有不少改进空间的。虽然对于是否会有人乐意使用“反常识”的逆波兰表达式写程序存有疑问，不过，改进上述语言，并应用于对于“堆栈”这种数据结构的教学，或许也不失为堆栈式语言一种存在价值。

综上，本次大作业，我们将参考PostScript，构建一种堆栈式语言。该语言命名为StackBOOM。

1.2 设计目标

本次大作业设计目标为仿照PostScript语言，设计一种“幼儿态”的堆栈式语言StackBOOM，其包含简单的数据处理能力，能够实现条件判断以及循环操作，并能够让用户定义自己的操作。

之所以称之为“幼儿态”，原因在于受限于作业时间限制以及个人能力等，本次设计的语言并没有实现所有目前主流语言的全部功能，例如生成字典这种数据结构。

对于期望StackBOOM实现的功能列举如下：

1. 能够实现数据的直接输入
2. 能够进行运算操作
3. 能够组合各种操作
4. 能够实现判断和循环
5. 能够部分实现并行

2. 词法设计

2.1 关键字

```
NEWSTACK = "news";
ENDSTACK = "ends";
WHILE = "while";
BREAK = "break";
CONTINUE = "continue";
IF = "if";
IFELSE = "ifelse";
DEF = "def";
POP = "pop";
PRINT = "print";
RETURN = "return";
KEYWORD = WHILE | FOR | IF | IFELSE | DEF ;
```

2.2 运算符

```
OPERATOR = "+" | "add" | "-" | "sub" | "*" | "mul" | "\" | "div" | "e" | "**" |
"pow" | "sqrt" | "%" | "mod" | PRINT | POP | RETURN | CONTINUE | BREAK;
JUDGE = "==" | "!=" | ">" | "<" | ">=" | "<=";
LOGIC = "&" | "and" | "|" | "or";
POPMARK = ",";
SAVEMARK = ";";
MARK = POPMARK | SAVEMARK;
```

2.3 标识符

```
LETTER = a | b | c | ... | z;
ALPHA = A | B | C | ... | Z;
NUM = 0 | 1 | 2 | ... | 9;
NAME = (LETTER | ALPHA | "_"), {LETTER | ALPHA | "_" | NUMBER};
OPNAME = "/", NAME;
VARNAME = "//", NAME;
```

2.4 数字与字符

```
SPACE = " ";
EOL = "/n";
INTEGER = ["-"], NUM, {NUM};
FLOAT = INTEGER, ".", NUM, {NUM};
NUMBER = INTEGER | FLOAT;
```

3. 语法设计

3.1 语句与语句块

```
VARDEF = VARNAME, SPACE, NUMBER, SPACE, DEF, EOL;
OPDEF = OPNAME, SPACE, NUM, SPACE, BLOCK, DEF, EOL;
VARDEFBLOCK = VARDEF | (VARDEF, VARDEFBLOCK);
OPDEFBLOCK = OPDEF | (OPDEF, OPDEFBLOCK);
DEFBLOCK = VARDEFBLOCK | OPDEFBLOCK;
OP = OPNAME | OPERATOR;
```

```
ITEM = VARNAME | NUMBER;
OPITEM = (ITEM, POPMARK) | (ITEM, SAVEMARK);
OPLIST = OPITEM | (OPITEM, SPACE, OPLIST);
OPLINE = (OP, EOL) | (OPLIST, SPACE, OP, EOL);
OPLINES = OPLINE | (OPLINE, EOL, OPLINES);
OPBLOCK = OPLINE | ("{", OPLINES, "}", EOL);
JUDGEITEM = OPBLOCK | (ITEM, SPACE);
JUDGELINE = JUDGEITEM, JUDGEITEM, JUDGE, EOL;
MULTIJUDGELINE = JUDGELINE | (JUDGELINE, SPACE, LOGIC, SPACE, MULTIJUDGELINE);
JUDGEBLOCK = JUDGELINE | ("{", JUDGELINE, LOGIC, SPACE, MULTIJUDGELINE, "}",
EOL);
IFBLOCK = OPBLOCK, JUDGEBLOCK, IF, EOL;
IFELSEBLOCK = OPBLOCK, OPBLOCK, JUDGEBLOCK, IFELSE, EOL;
WHILEBLOCK = OPBLOCK, JUDGEBLOCK, WHILE, EOL;
CALBLOCK = OPBLOCK | JUDGEBLOCK | IFBLOCK | IFELSEBLOCK | WHILEBLOCK;
BLOCK = DEFBLOCK | CALBLOCK | (BLOCK, BLOCK);
```

3.2 整体程序

```
PROGRAM = STACK;
STACK = (NEWSTACK, EOL, STACK_BLOCK, EOL, ENDSTACK) | (STACK, EOL, STACK);
STACK_BLOCK = STACK | BLOCK | (STACK_BLOCK, STACK_BLOCK);
```

4. 语义说明

4.1 存储域 & 辅助函数

内存
Store = Location → (stored Storable + undefined + unused + marked) // marked标记地址内容用了savemark标记

- 语义函数

```
empty_store : Store
allocate : Store → Store × Location
deallocate : Store × Location → Store
update : Store × Location × Storable → Store
fetch : Store × Location → Storable
push : stble -> Stack
```

- 语义

```
empty_store = λloc.unused
allocate sto =
    let loc = any_unused_location (sto) in
    (sto [loc→ undefined], loc)
deallocate (sto, loc) = sto [loc → unused]
update (sto, loc, stble) = sto [loc→stored stble]
fetch (sto, loc) =
    let stored_value (stored stble) = stble
        stored_value (undefined) = fail
        stored_value (unused) = fail
    in stored-value (sto(loc))
push stble
```

4.2 环境域 & 辅助函数

```
Environ = Identifier → (bound Bindable + unbound)
● 语义函数
empty_environ : Environ
bind : Identifier×Bindable → Environ
overlay : Environ×Environ → Environ
find : Environ×Identifier→Bindable

● 语义
enpty-environ = λI. unbound
bind (I，bdble) = λI'. if I'=I then bound bdble else unbound
overlay (env'，env) = λI. if env' (I)/=unbound then env' (I) else env (I)
find (env，I) =
    let bound_value (bound bdble) = bdble
        bound_value (unbound) = ⊥
    in bound_value (env (I))
```

4.3 语义域

```
● 值
number = integer Integer + float Float
Value = truth_value Truth_Value + number Number
VAL_LENTH : 常量，为一个NUMBER所需的地址字节数
● 堆栈
Stack = Location × Location //栈起始地址 × 当前地址
● 辅助函数
next_loc Stack → Stack
prev_loc Stack → Stack

next_loc sta = sta[loc1 × (loc2 + VAL_LENGTH)]
prev_loc sta =
    let (loc1, loc2) = sta in
    if (loc2 - VAL_LENGTH < loc1) then error
    else sta[loc1 × (loc2 - 1)]

● 语义函数
get_new_stack : Store → Stack //在内存中创建新的堆栈
push : Stack × Store × Number → Stack × Store //将Number存入堆栈Stack
pop : Stack × Store → Stack × Store × Number //将取出Stack顶部的Number
stack_length : Stack → Integer //返回栈的长度
stack_top : Stack → Location //返回当前栈顶

● 语义
get_new_stack sto =
    let (sto, loc) = allocate(sto) in
    sta[loc × loc]
push sta sto val =
    let sta2[loc1, loc2] = next_loc(sta) in
    (sta2, update(sto, loc2, val))
pop val =
    let (loc1, loc2) = sta in
```

```
    let val = fetch(sto, loc2) in
    let sta2 = prev_loc(sta) in
    (sta2, sto, val)
stack_length sta =
    let (loc1, loc2) = sta in
    (loc2 - loc1) / VAL_LENGTH
stack_top sta =
    let (loc1, loc2) = sta in
    loc2
```

4.4 指称语义

4.4.1 整体程序

```
PROGRAM = STACK;
STACK = (NEWSTACK, EOL, STACK_BLOCK, EOL, ENDSTACK) | (STACK, EOL, STACK);
STACK_BLOCK = STACK | BLOCK | (STACK_BLOCK, STACK_BLOCK);
```

- 语义函数

```
run: PROGRAM → (Environ → Store → Store)
run_stack: STACK → (Environ → Store → Store)
run_sta_blo = (STACK + BLOCK) → (Environ × Stack → Store → Environ × Stack ×
Store)
execute: BLOCK → (Environ × Stack → Store → Environ × Stack × Store)
```

- 语义

```
run [stack] env sto = run_stack [stack] env sto

run_stack [news sta_blo ends] env sto =
    let newstack = get_new_stack(sto) in
    let (env', sta', sto') = run_sta_blo sta_blo env newstack sto in
    (sto')
run_stack [stack1 stack2] env sto =
    run_stack stack2 env (run_stack stack1 env sto)

run_sta_blo [stack] env sta sto =
    (env, run_stack [stack] env sto, sta)
run_sta_blo [block] env sta sto =
    execute [block] env sta sto
run_sta_blo [sta_blo1 sta_blo2] env sta sto =
    run_sta_blo [sta_bloc2] (run_sta_blo sta_blo1 env sta sto)
```

4.4.1 语句块

```
BLOCK = DEFBLOCK | CALBLOCK | (BLOCK, BLOCK);
```

- 语义函数

```
execute: BLOCK → (Environ × Stack → Store → Environ × Stack × Store)
elaborate: DEFBLOCK → (Environ × Stack → Store → Environ × Store)
cal_block : CALBLOCK → (Environ × Stack → Store → Stack × Store)
```

- 语义

```
execute [defblock] env sta sto =
```

```
    let (env', sto') = elaborate [defblock] env sta sto in
    (env', sta, sto')

execute [calblock] env sto sta =
    (env, cal_block [calblock] env sta sto)

execute [block1 block2] env sto sta =
    execute block2 (execute block1 env sto sta)
```

4.4.1.1 定义语句块

```
VARDEF = VARNAME, SPACE, NUMBER, SPACE, DEF, EOL;
OPDEF = OPNAME, SPACE, NUM, SPACE, BLOCK, DEF, EOL;
VARDEFBLOCK = VARDEF | (VARDEF, VARDEFBLOCK);
OPDEFBLOCK = OPDEF | (OPDEF, OPDEFBLOCK);
DEFBLOCK = VARDEFBLOCK | OPDEFBLOCK;

● 语义函数
elaborate: DEFBLOCK → (Environ × Stack → Store → Environ × Store)

● 语义函数
elaborate [//a N def] env sta sto =
    let (sto', loc)= allocate sto in
    (bind (//a, N loc), sto')
elaborate [vardef1 vardefblock] env sta sto =
    let (env', sto') = elaborate vardef1 env sta sto in
    elaborate vardefblock env' sta sto'

elaborate [OP NUM BLOCK def] env sta sto =
    let operation =
        let sta_len = stack_length(sta) in
        if sta_len < NUM then error
        else
            execute BLOCK env sto sta
    in
    (bind (OP, operation), sto)
elaborate [opdef1 opdefblock] env sta sto =
    let (env', sto') = elaborate opdef1 env sta sto in
    elaborate opdefblock env' sta sto'
```

4.4.1.2 变量操作

```
VARNAME = "//", NAME;
NUMBER = INTEGER | FLOAT;
ITEM = VARNAME | NUMBER;
OPITEM = ITEM POPMARK | ITEM SAVEMARK;

● 语义函数
dealitem : OPITEM → (Environ × Stack → Store → Stack × Store)

● 语义
dealitem [//a,] env sta sto=
    let loc = find (env, //a) in
    let val = fetch(sto, loc) in
```

```
    push sta val sto
dealitem [//a;] =
    let loc = find (env, //a) in
    let val = fetch(sto, loc) in
    let (sta', sto') = push sta val sto in
    let loc = stack_top sta' in
    store[loc → marked]
dealitem [num,] =
    push sta num sto
dealitem [mum;] =
    let (sta', sto') = push sta num sto in
    let loc = stack_top sta' in
    store[loc → marked]
```

4.4.1.3 操作语句块

```
OP = OPNAME | OPERATOR;
ITEM = VARNAME | NUMBER;
OPITEM = (ITEM, POPMARK) | (ITEM, SAVEMARK);
OPLIST = OPITEM | (OPITEM, SPACE, OPLIST);
OPLINE = (OP, EOL) | (OPLIST, SPACE, OP, EOL);
OPLINES = OPLINE | (OPLINE, EOL, OPLINES);
OPBLOCK = OPLINE | ("{", OPLINES, "}", EOL);
```

- 语义函数

```
do : OPLINE → (Environ × Stack → Store → Stack × Store)
do_lines : OPLINES → (Environ × Stack → Store → Stack × Store)
cal_block : CALBLOCK→ (Environ × Stack → Store → Stack × Store)
deal_item_list : OPLIST → (Environ × Stack → Store → Stack × Store)
calculate : OPERATOR → (Environ × Stack → Store → Stack × Store)
```

- 语义

```
do [ope] env sta sto =
    calculate [ope] env sta sto
do [opn] env sta sto =
    let op = find (env, opn) in
    op env sta sto
do [opl ope]
    let (sta', sto') = deal_item_list opl env sta sto in
    calculate [ope] env sta' sto'
do [opl opn]
    let (sta', sto') = deal_item_list opl env sta sto in
    let op = find (env, opn) in
    op env sta' sto'

do_lines [opl] env sta sto =
    do opl env sta sto
do_lines [opl opls] env sta sto =
    do_lines opls env (do opl env sta sto)

cal_block [opl] env sta sto =
    do opl env sta sto
cal_block [{ \n opls \n }] env sta sto =
    do_lines opls env sta sto
```

```
deal_item_list [opi] env sta sto =
    dealitem opi env sta sto
deal_item_list [opi opl] env sta sto =
    deal_item_list opl env (dealitem opi env sta sto)
```

4.4.1.4 表达式计算

```
OPERATOR = "+" | "add" | "-" | "sub" | "*" | "mul" | "\" | "div" | "e" | "**" |
"pow" | "sqrt" | "%" | "mod" | PRINT | POP | RETURN | CONTINUE | BREAK;

● 语义函数
calculate : OPERATOR → (Environ × Stack → Store → Stack × Store)

● 语义
calculate [+] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 + x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2') //如果有标量被标记为makred，则将值存入对应的
内存
calculate [add] env sta sto = calculate [+] env sta sto

calculate [-] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 - x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')
calculate [sub] env sta sto = calculate [-] env sta sto

calculate [*] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 * x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')
calculate [mul] env sta sto = calculate [*] env sta sto

calculate [\] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
```

```
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 + x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')
calculate [div] env sta sto = calculate [\] env sta sto

calculate [**] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 ** x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')
calculate [pow] env sta sto = calculate [**] env sta sto

calculate [sqrt] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = sqrt(x1, x2) in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')

calculate [%] env sta sto =
    let (loc1, loc2) = sta in
    let (sta', sto', x1) = pop sta sto in
    let (loc1', loc2') = sta' in
    let (sta'', sto'', x2) = pop sta' sto' in
    let result = x1 % x2 in
    let push sto sta result in
    (λI'.if sto[I' → marked] then
        sto[I' → storable]
        update sto I' result) (loc2 loc2')
calculate [mod] env sta sto = calculate [%] env sta sto

calculate [print] env sta sto =
    let (sta', sto', x1) = pop sta sto in
    print x1

calculate [pop] env sta sto = pop sta sto
```

4.4.1.5 条件判断

```
JUDGEITEM = OPBLOCK | (ITEM, SPACE);
JUDGELINE = JUDGEITEM, JUDGEITEM, JUDGE, EOL;
MULTIJUDGELINE = JUDGELINE | (JUDGELINE, SPACE, LOGIC, SPACE, MULTIJUDGELINE);
```

```
JUDGEBLOCK = JUDGELINE | ("{", JUDGELINE, LOGIC, SPACE, MULTIJUDGELINE, "}",
EOL);
```

- 语义函数

```
get_result : JUDGEITEM → (Environ × Stack → Store → Value)
judge : JUDGELINE → (Environ × Stack → Store → Value)
judge_muilt : MULTIJUDGELINE → (Environ × Stack → Store → Value)
cal_block : CALBLOCK → (Environ × Stack → Store → Value)
```

- 语义

```
get_result [opblock] env sta sto =
    let (sta', sto') = cal_block opblock env sta sto in
    let (sta'', sto'', val) = pop sta' sto' in
    val
get_result [//a] env sta sto =
    let loc = find (env, //a) in
    fetch(sto, loc)
get_result [number] env sta sto = number

judge [jitem1 jitem2 ==] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 == v2 then true else false
judge [jitem1 jitem2 !=] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 != v2 then true else false
judge [jitem1 jitem2 <] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 < v2 then true else false
judge [jitem1 jitem2 >] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 > v2 then true else false
judge [jitem1 jitem2 <=] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 <= v2 then true else false
judge [jitem1 jitem2 >=] env sta sto =
    let v1 = get_result jitem1 env sta sto in
    let v2 = get_result jitem2 env sta sto in
    if v1 >= v2 then true else false

judge_muilt [jline] env sta sto = judge [jline] env sta sto
judge_muilt [jline == jmuilt] =
    let v1 = judge jline1 env sta sto in
    let v2 = judge_muilt jmuilt env sta sto in
    if v1 >= v2 then true else false
```

4.4.1.6 循环和条件块

```
IFBLOCK = OPBLOCK, JUDGEBLOCK, IF, EOL;
IFELSEBLOCK = OPBLOCK, OPBLOCK, JUDGEBLOCK, IFELSE, EOL;
WHILEBLOCK = OPBLOCK, JUDGEBLOCK, WHILE, EOL;
```

- 语义函数

```
cal_block : CALBLOCK → (Environ × Stack → Store → Stack × Store)
```

- 语义

```
cal_block [block jblock if] env sta sto =
    let v1 = cal_block jblock env sta sto in
    if v1 then do_bock block env sta sto

cal_block [block1 block2 jblock ifelse] env sta sto =
    let v1 = cal_block jblock env sta sto in
    if v1 then do_bock block1 env sta sto
        else do_bock block2 env sta sto

cal_block [block jblock while] env sta sto =
    let do_while env sta sto =
        let v1 = cal_block jblock env sta sto in
        if v1 then
            do_while env (cal_block block env sta sto)
        else (sta, sto)
    in
    do_while
```

5. 与对标语言的差异

其实本语言仅仅是模仿了PostScript部分语法格式，但实际上这部分重合的语法格式仅仅是来源于逆波兰表达式，也就是操作符后缀。PostScript语言主要用于文字排版，使得其大量定义的操作都是与尺寸、换行、重复、字符操作有关；而本语言则是设计得更像是计算器一样，更偏重于数字操作。并且本语言强化了运行过程中“栈”的概念，通过增加是否出栈标识符，希望使用者能够更加自如、至少是更加明确地定义对数据的出栈操作。在此种意义上，本语言确实是将数据提升为了一等公民。

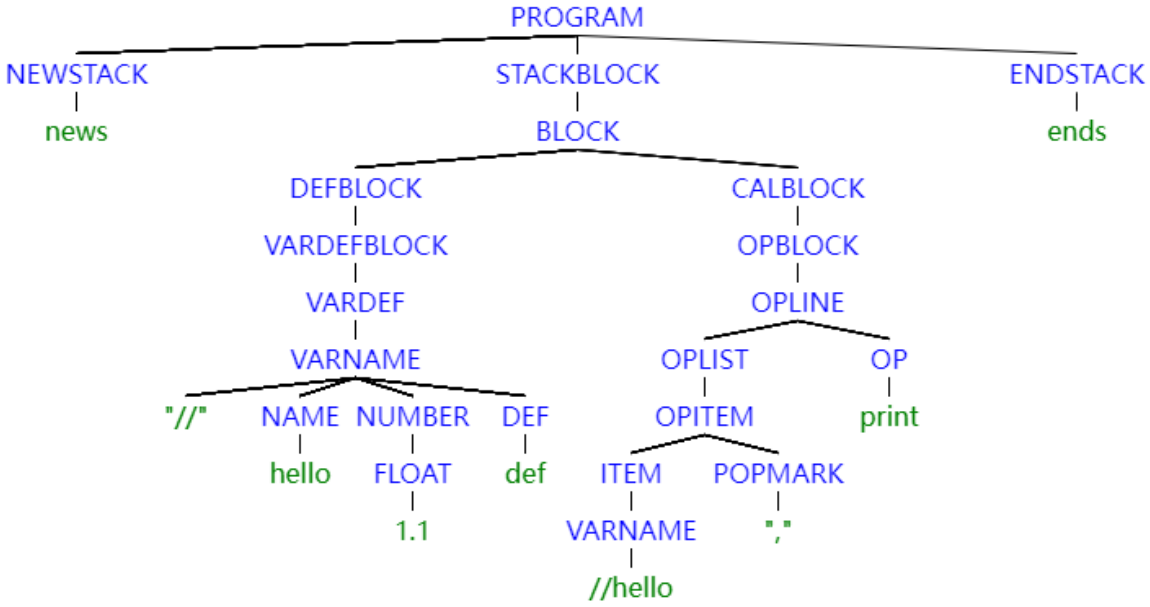
6. 部分用法示例

6.1 Hello World

由于本语言没有提供字符串的处理，在此仅输出数字：

```
news
    //hello 1.1 def
    //hello, print
ends
```

上述程序预计输出“1.1”。
对其绘制语法树如下：



6.2 基础程序示例

下面的程序可以实现斐波那契数列前13项的打印输出：

```
news
    //a 1 def
    //b 1 def
    //i 0 def
    //a, print
    //b, print
    {
        {
            //a; //b, +
            print
        }
        {
            //b; //a, add
            print
        }
        {
            //i, 2, %
            0 ==
        }
        ifelse
        //i; 1, +
        pop
    }
    {
        //i 10 <=
    }
    while
ends
```

下面进行一些说明：

- 对于每一个值，例如上述程序中的`//a`、`1`等，在出现时相当于一次入栈操作；

- 对于每一个操作，如`def`、`add`等，他们有预定义完全的需要的参数数量以及类型，例如`add`的操作需要两个数（整数或浮点数），那么他会从栈顶取出两个数（出栈），在他们类型正确的情况下进行运算，并将运算结果入栈；
- 对于`OPITEM`，末尾有标识，分以下几种情况：
    - 标识前为数或字符串，标识为`SAVEMARK`：该值会按顺序留存在栈中，不受操作出栈的影响；
    - 标识前为数或字符串，标识为`POPMARK`：该值会受操作出栈的影响，运算后不再存储在栈中；
    - 标识前为变量名，标识为`SAVEMARK`：该变量对应值会在操作时出栈；操作得到的结果会在入栈的同时存储到对应变量名下；
    - 标识前为变量名，标识为`POPMARK`：该变量对应值会在操作时出栈。

6.3 函数定义

下面程序将6.1中的循环改变为递归：

```
news
    /once 3 {
        //a 0 def
        //b 0 def
        //a; add
        add
        //b; add
        print
        //i 0 def
        //i; add
        return
        {
            //i, 1 add
            //a, //b, /once
        }
        //i 10 >
        ifelse
    }
    def
    0, 1, 1, /once
ends
```

对于上述程序进行部分解释：

- `/once`为函数名称，后面的`3`表示函数调用时，栈中至少有三个值（数字或字符串），也就是说函数有三个参数；
- 因为语言没有为函数参数进行命名，因而只能用顺序来确定各参数意义。`//a`用于取出并存储栈顶部的参数，取出办法为初始化`//a`为0，并使用`//a; add`来将0与栈顶值相加（因为`add`操作会取出栈顶两个值进行运算）；因为`;`的存在，`add`的结果被存放到栈顶后，同时会对`//a`进行赋值。
- `//b`用于存储传入的栈顶的两个参数的加法运算结果。
- 传入的第三个参数意义为6.1中的`//i`，对它的值进行读取，并判断是否结束递归。

6.4 多线程

本语言的`NEWSTACK`和`ENDSTACK`中间囊括的代码段运行时使用了“同一个栈”，换句话说可以认为该段程序运行在一个线程上。因此，`NEWSTACK`和`ENDSTACK`关键字并不是像其他语言中大括号或者`Begin`、`End`等符号一样，仅仅指示一段程序的开始与结束，而是可以指示多个不同线程的代码段。
这里将不再使用完整程序进行示例。