An Interpreter for Three-Address Code

1. Three-Address Code

1.1. Grammar

The grammar for the three-address code instructions is given as follows (nonterminals in italics and terminals in boldface) :

program → *procedures* *main*
procedures → *procedure* *procedures* | ε
procedure → **procstart** *stmts* **procexit**
main → **start** *stmts* **exit**
stmts → *stmt* *stmts* | ε
stmt → *label_stmt* | *IO_stmt* | *assign_stmt* | *jump_stmt* | *proc_stmt*
label_stmt → **id**
IO_stmt → **readint** *var* | **writeint** *factor* | **readchar** *stack_var* | **writechar** *factor*
assign_stmt → *stack_var* **:=** *expr* | * *stack_var* **:=** *factor*
jump_stmt → **goto** **id** | **if** *factor* *relop* *factor* **goto** **id**
proc_stmt → **setup_call** **const** | **param** *factor* | **call** **id** | **return** | **return** *factor* | **retrieve** *stack_var*
stack_var → **sp [** *signed_const* **]**
expr → *factor* *binop* *factor* | – *factor* | *factor* | * *stack_var*
factor → *stack_var* | **intconst** | **sp** | **charconst**
binop → **+** | **–** | ***** | **/**
relop → **<** | **<=** | **>** | **>=** | **=** | **<>**
signed_const → **const** | **– const**

1.2. Lexical Conventions

id → letter (letter | digit | _)*
const → positive integers

1.3. Statement Format

(1) Only one statement may appear on a line.
(2) **Procstart, procexit, start,** and **exit** are reserved labels.
(3) Tabs and blanks may be used to increase human readability.
(4) Comment lines are denoted by a ‘**#**’ in the first column.

2. Stack Environment

An activation record has following layout :

…	temporaries
sp[m]	local_m
…	…
sp[1]	local_1
sp→	return address
sp[-1]	caller's sp
sp[-2]	*n*th parameter
…	…
sp[-(n+1)]	lst parameter
sp[-(n+2)]	return value

Stack grows from low address to high address.

The instruction **setup_call n** signals the first stack location (relative to the current stack pointer) at which parameters will be placed. The number **n** is the amount of space allocated so far to the locals and temporaries in the current activation record.

Example : The current environment has 2 local variables and 3 temporary variables, and needs to call the function **bar** with 2 actual parameters in temp1 and temp2. The instruction sequence

```
setup_call    5
param         temp1
param         temp2
call          bar
retrieve      sp[6]
```

will place the first parameter at sp[7] (i.e., skip 6 words) and the second parameter at sp[8], then transfer control to **bar.** After returning from **bar**, the return value can be accessed at sp[6].

3. Intermediate Code Generation

3.1. Referencing Parameters

Parameters are referenced by displacing off the stack pointer **sp**, e.g., if **x** is a parameter at displacement -4 from **sp** then it can be referenced as

sp[-4] := …

The *i*th element of a parameter array **A** can be referenced as follows: Let **A** be at displacement -4 from **sp** and the value of *i* have been computed into a temporary **temp1.** Then, we can generate the following code:

temp2 := sp[-4] + temp1

temp3 := *temp2

This results in the value of **A[** *i*] being loaded into **temp3.** Suppose we want to store the value in **temp0** into **A**[*i*] **.** We can generate the following code :

temp2 := sp[-4] + temp1

***temp2 := temp0**

A field of a parameter object can be accessed similarly by directly adding the offset of the field in the object and the displacement of the object.

3.2. Referencing Locals

Local variables are referenced by displacing off the stack pointer **sp**, e.g., if **x** is a local variable at displacement 4 from **sp** then it can be referenced as

sp[4] := …

The *i*th element of a local array **A** starting at displacement **D** from **sp** (i.e., its first element is at **sp**[**D**]) can be referenced as follows: Let the value of *i* has been computed into a temporary **sp**[**k**] . Then, we can generate the following code (note that **D** is known at compile time) :

```
sp[n1]  :=  sp[k] + D      /* displacement of i^th element */
sp[n2]  :=  sp + sp[n1]    /* location of i^th element */
sp[n3]  :=  *sp[n2]        /* content of i^th element */
```

where **n1, n2** and **n3** are displacements of local temporaries. Suppose we want to store the value in temporary **sp**[**n0**] into the *i*th element of **A** . Then, we can generate the following code :

```
sp[n1]  :=  sp[k] + D      /* displacement of i^th element */
sp[n2]  :=  sp + sp[n1]    /* location of i^th element */
*sp[n2] :=  sp[n0]
```

4. Input and Output

The instructions **readint**, **writeint**, **readchar**, and **writechar** perform input and output. The instruction

readint x

reads an integer and stores it at the location **x.** The instruction

writeint x

writes the integer stored at the location **x**. The instruction

readchar x

reads a character and stores it's ASCII code at the location **x.** The instruction

writechar x

writes the character with its ASCII code stored at the location **x**.

5. An Example

The following source program:

```
class foo {
    int z;
    int bar(int x[], int y)
    {
        z = x[0]+x[1]*y;
        return z;
    }
}

class Program {
    void main( )
    {
        foo obj;
        int a[2];
        int b;

        a[0] = 0;
        a[1] = 1
        b = readint( );
        writestring("bar=");
        writeint( obj.bar( a, b ) );
        b = b * 3 + a[0];
        writeint( obj.bar( a, b ) );
    }
}
```

Could possibly be translated into:

```
procstart
foo_bar
#   this    @ sp[-4] ,    x    @ sp[-3] ,    y    @ sp[-2]
#   t1      @ sp[1] ,     t2   @ sp[2]
sp[1]   :=   sp[-3] + 0
sp[2]   :=   *sp[1]
sp[3]   :=   sp[-3] + 1
sp[4]   :=   *sp[3]
sp[5]   :=   sp[4] * sp[-2]
sp[6]   :=   sp[2] + sp[5]
sp[7]   :=   sp[-4] + 0
*sp[7]   :=   sp[6]
sp[8]   :=   sp[-4] + 0
sp[9]   :=   *sp[8]
return   sp[9]
procexit

start
#   obj    @ sp[1] ,    obj.z    @ sp[2]
#   a      @ sp[3] ,    a[0]    @ sp[4] ,    a[1]    @ sp[5],    b   @ sp[6]
#   store the starting address of obj and a
sp[1]   :=   sp + 2
sp[3]   :=   sp + 4

sp[4]   :=   0
sp[5]   :=   1
readint        sp[6]
writechar      ‘b’
writechar      ‘a’
writechar      ‘r’
writechar      ‘=’

#   sp[7]   :=   obj.bar( a, b )
#   sp[1] ~ sp[6] are used
setup_call     6
param          sp[1]
param          sp[3]
param          sp[6]
```

```
call            foo_bar
retrieve        sp[7]
writeint        sp[7]

#   sp[7] and sp[8] are temporary variables used after previous call
sp[7]   :=   sp[6] * 3
sp[8]   :=   sp[7] + sp[4]
sp[6]   :=   sp[8]

#   sp[9]   :=   obj.bar( a, b )
#   sp[1] ~ sp[8] are used
setup_call      8
param           sp[1]
param           sp[3]
param           sp[6]
call            foo_bar
retrieve        sp[9]
writeint        sp[9]
exit
```