STACKL Reference Manual

1 Introduction 4

2 stacklc: the stackl compiler 4

2.1 Compiler modes 4

2.2 Special functions 4

2.2.1 Interrupt 5

2.2.2 system trap 5

2.2.3 startup 6

2.3 Generating assembly code 6

3 The stackl machine 7

3.1 Execution Registers 7

3.1.1 Instruction Pointer (IP) 7

3.1.2 Stack Pointer (SP) 7

3.1.3 Frame Pointer (FP) 7

3.2 Memory Management Registers 7

3.2.1 Base Pointer (BP) 8

3.2.2 Limit Pointer (LP) 8

3.3 Stack Frames 8

3.4 Input/Output 8

3.5 Loading programs 10

4 Devices 10

4.1 Programmed IO Terminal (pio\_term) 11

4.1.1 Registers 11

4.2 Timer 12

5 Interpreter Architecture 12

5.1 Memory 12

5.2 Executable file format and the Loader 13

5.3 Adding IO devices to the interpreter 13

6 Instruction Reference 13

6.1 ADJSP <amount> 13

6.2 AND 13

6.3 CALL <addr> 14

6.4 CALLI 14

6.5 CLID 14

6.6 DIVIDE 14

6.7 DUP 14

6.8 EQ 15

6.9 GE 15

6.10 GT 15

6.11 HALT 15

6.12 ILLEGAL 15

6.13 INP 15

6.14 JMPUSER <dest> 15

6.15 JUMPE <dest> 16

6.16 JUMP <dest> 16

6.17 LAND 16

6.18 LE 16

6.19 LOR 16

6.20 LT 16

6.21 LXOR 17

6.22 MINUS 17

6.23 MOD 17

6.24 NEG 17

6.25 NE 17

6.26 NOP 17

6.27 OR 17

6.28 OUTS 18

6.29 PLUS 18

6.30 POPCVARIND 18

6.31 POPCVAR <offset> 18

6.32 POPREG <reg num> 18

6.33 POPVAR <offset> 19

6.34 POP 19

6.35 PUSHCVARIND 19

6.36 PUSHCVAR <offset> 19

6.37 PUSHFP 19

6.38 PUSHREG <reg num> 19

6.39 PUSHVAR <offset> 20

6.40 PUSH <value> 20

6.41 RETURNV 20

6.42 RETURN 20

6.43 RTI 21

6.44 SEID 21

6.45 SETMODE <mode> 21

6.46 SHIFTL 21

6.47 SHIFTR 21

6.48 SWAP 21

6.49 TIMES 21

6.50 TRAPTOC 22

6.51 TRAP 22

Appendix A stacklc grammar 23

# Introduction

The stackl environment is intended as a virtual environment that can be used for teaching operating systems classes. The environment allows students to program in C and be able to write interrupt service routines, syscalls, and other operating system code without the need for real hardware.

# stacklc: the stackl compiler

The stacklc compiler compiles a subset of the C programming language. The following are the current restrictions:

1. No separate compilation
2. The switch statement is not implemented
3. void is not supported
4. static is not supported

The compiler will output an .sl file that contains the assembly version of the program. It will then invoke the assembler to create an .slb file that can be run by the interpreter.

## Compiler modes

The compiler runs in two modes: boot and regular. Boot mode is selected with the boot command line option. In regular mode, the compiler includes the file <startup.h> prior to compiling any other source code. It is assumed that the startup code will supply a startup\_\_ function that will call main() and then exit (halt the CPU). The startup code must also define an interrupt and systrap function, which are defined in the next section.

In boot mode, the compiler does not imply any include file. The compiled source must include definitions for the startup\_\_, systrap, and interrupt functions.

## Special functions

The stackl interpreter requires knowledge of three specific functions:

1. The interrupt service routine
2. The system trap routine
3. The startup routine

These can use any name. They are identified to the compiler via pragmas.

### Interrupt

The interrupt routine is called when any hardware interrupt occurs. It takes no arguments and must not declare any local variables. If local variables are required, the interrupt routine can call another routine. The called routine does not have any restrictions.

The interrupt routine must return using the RTI instruction. A sample interrupt routine is shown below:

int process\_interrupt()

{

int local\_variables\_are\_ok;

// process interrupt

return 0;

}

#pragma interrupt isr

int isr()

{

process\_interrupt();

asm("RTI");

}

If a compilation unit does not have the pragma for the interrupt, the compiler will place an invalid address into the vector. If a hardware interrupt occurs, a machine check will result.

### system trap

The system trap routine is called when the TRAP instruction executes. It must not declare any local variables. If local variables are required, the interrupt routine can call another routine. The called routine does not have any restrictions.

The system trap routine must return using the RTI instruction.

The arguments of the function that executed the TRAP will be available to the system trap function. Normally, the TRAP is contained within a syscall function. This setup facilitates passing information to the system trap function

A sample system trap routine is shown below:

int syscall(int op, char \*buffer)

{

asm("TRAP");

return 0;

}

#pragma systrap systrap

int systrap(int op, char \*buffer)

{

if (op == PRINT) asm("OUTS", buffer);

asm("RTI");

}

If a compilation unit does not have the pragma for the system trap, the compiler will place an invalid address into the vector. If a TRAP instruction is executed, a machine check will result.

### startup

The startup routine is where the code begins executing. If a compilation unit does not have a pragma for the startup routine, the compiler will generate the following startup routine:

#pragma startup \_\_startup

int \_\_startup()

{

main();

asm("HALT");

}

## Generating assembly code

Assembly instructions can be generated using the asm and asm2 keywords. The asm and asm2 keywords behave like functions with the following prototype:

int asm(opcode, ...);

int asm2(opcode, opcode2, ...);

The optional parameters will be pushed onto the stack, then the opcode value(s) will be emitted to the instruction stream. The asm2 form is useful for two word instructions such as PUSH or JMPUSER.

If the asm keywords are used in an assignment statement, then the value at the top of the stack after the opcode executes will be popped and stored in the lval of the assignment.

Note that the asm instructions cannot be used in an expression. They can be used as a stand-alone statement, or as the only element on the right side of an assignment.

# The stackl machine

The stackl machine is a stack based machine. As such, there are no general purpose registers. All operations make use of the stack. Typically, the operands of an opcode are popped off the stack and the result of the operation is pushed back onto the stack.

There are two categories of special purpose registers: execution and memory management. The execution registers are used for general instructions. The memory management registers are used to manage memory when multiple processes are running on the machine. The registers are described in the following sections.

## Execution Registers

### Instruction Pointer (IP)

The instruction pointer contains the address of the next instruction to be executed.

### Stack Pointer (SP)

The stack pointer contains the address of the first unused memory location at the top of the stack. The stack always contains word aligned data. The stack grows to increasing memory addresses.

Pseudocode for pushing an item onto the stack is as follows:

Memory[SP] = value;

SP++;

Pseudo-code for popping an item from the stack is as follows:

SP--;

value = Memory[SP];

### Frame Pointer (FP)

The frame pointer is used for function calls. It points to the location of the first local variable in a called function. The opcodes that include VAR in the opcode name reference the FP to locate the variable.

## Memory Management Registers

When operating in user mode, memory accesses are relative to the Base Pointer. When not in user mode, the Base Pointer and Limit Pointer are ignored, and all addresses are absolute.

### Base Pointer (BP)

The base pointer contains the address of the start of memory for the current process. All memory addresses are relative to the Base Pointer. In other words, if a process wants to access memory at address N, the actual address is BP+N. This is true whether the memory operation is fetching data or instructions.

### Limit Pointer (LP)

The limit pointer contains the value just above the highest memory that a process can access. In other words, legal memory access are those in the range of BP .. LP-1. Any access outside this range will cause a machine check.

## Stack Frames

The following diagram illustrates a stack frame. Higher addresses are higher in the diagram.



The caller must push the parameters onto the stack prior to executing the CALL instruction. Parameters must be pushed from right to left. In other words, a call to

foo(a,b,c)

must push c then b then a.

## Input/Output

The CPU has two instructions for doing input/output the OUTS instruction prints the NULL terminated string whose address is on the top of the stack. This is a blocking instruction: the next instruction will not execute until the IO operation is complete.

The INP instruction provides a non-blocking means of doing input. The value on the top of the stack is assumed to be the address of an io\_blk\_t. The IO block has the following form:

typedef struct

{

int op;

int param1;

int param2;

} io\_blk\_t;

The op is used to specify the desired operation and indicates the completion of the operation. The two param values are operation dependent.

When the operation is finished the most significant bit of the op field is set (resulting in a negative value).

The following operations are defined:

GETL\_CALL 6 This operation will perform a C gets() function. The address of the buffer is in param1. This function will read at most 255 characters plus a terminating null.

GETS\_CALL 5 This operation will perform the following C function:

scanf("%s", param1);

where param1 is the address of the buffer. There is no attempt to prevent buffer overflow.

GETI\_CALL 7 This operation will perform the following C function:

scanf("%d", param1);

where param1 is assumed to be the address of an integer.

EXEC\_CALL 8 This operation will call the loader to load a file into memory. The name of the file is in a string pointed to by param1. The high memory address of the load will be stored in param2 upon completion. The BP and LP registers should be set prior to initiating this operation with the BP indicating the starting address for the load.

Since these are non-blocking operations, the initiator needs to poll the op field waiting for it to turn negative to determine when the operation is complete.

The INP instruction is not enabled by default. To enable the INP instruction use the following pragma:

#pragma feature inp

## Loading programs

Programs can be loaded into memory using the INP instruction with the EXEC\_CALL operation. The loader will load the code starting at the address indicated by the BP. The loader will adjust addresses so that they are relative to BP. After loading, the param2 of the io\_blk will contain the address of the first free word above the loaded file. This value can be used to set the SP and FP for executing the code.

The following procedure should be used to load code:

1. set the BP and LP to bound the memory available to the program
2. initiate the load using the INP instruction
3. wait for the load to complete
4. Clear the interrupt mode bit in the FLAG register (if set)
5. Set the FP and SP to the value in param2
6. Use the JMPUSER instruction to jump to address 8

Note that this is address 8 in the user’s code (relative to the BP), it is not absolute address 8.

The code for doing this is as follows:

// set BP to start\_addr

asm2("POPREG", BP\_REG, start\_addr);

// initiate and wait for INP instruction

io\_blk.op = EXEC\_CALL;

io\_blk.addr = "ostest/user.slb";

io\_blk.status = 0;

asm("INP", &io\_blk);

while((io\_blk.op & IO\_COMPLETE) == 0)

{

}

// start the user code

asm2("POPREG", SP\_REG, io\_blk.param2); // update SP

asm2("POPREG", FP\_REG, io\_blk.param2); // update FP

asm2("JMPUSER", 8); // jump to 8 in user mode

Following the update to the FP, no VAR instructions should execute because they would reference the new FP. Following the update to the SP, stack operations will reference the new stack.

# Devices

This section discusses the IO devices available on the stackl machine.

## Programmed IO Terminal (pio\_term)

This device is used for doing programmed IO from/to the console. The device can operate in polled mode or in interrupt mode.

The pio\_term must be enabled using the following pragma:

#pragma feature pio\_term

### Registers

The following registers are available for this device. These are byte-wide registers and should only be accessed using byte wide instructions. If a word sized instruction is used to read or write these registers, the behavior is undefined.

#### RDR – Read Data Register

This register is at address 0x0E000000. This is a read only register.

Data input from the console will appear in this register. Every time a new value appears in this register, the RECV bit in the IIR register will be set. The bit will be cleared when the data is read from the RDR.

#### XDR – Xmit Data Register

This register is at address 0x0E000000. This is a write only register.

Data written to this register will be sent to the console. Writing to this register will clear the XMIT bit in the IIR. The XMIT bit in the IIR will be set when the device is ready to transmit another byte.

#### IER – Interrupt Enable Register

This register is at address 0x0E000001. This is a read/write register.

This register controls what events will trigger an interrupt. Interrupts will be triggered if the bit corresponding to the event is set in this register. Bit 1 controls the READ interrupt, bit 2 controls the XMIT interrupt.

#### IIR – Interrupt Information Register

This register is at address 0x0E000002. This is a read only register. Reading the register will clear it.

This register indicates what caused the interrupt. The following bits are used:

0x01 ATTN: This device needs attention

0x02 RECV: The device received a new character

0x04 XMIT: The device finished transmitting a byte

## Timer

A timer can be configured to generate an interrupt every N instructions. The timer has three word-sized registers. If the registers are accessed using byte instructions, the behavior is undefined.

The TIMER\_CSR register is at address 0x0C000000. Bit zero is the interrupt enable bit. Setting it to 1 will cause the timer to generate interrupts when the TIMER\_COUNT register is greater than or equal to the TIMER\_LIMIT register. Bit 31 is set when an interrupt is generated. Bit 31 is cleared by reading the TIMER\_CSR.

The TIMER\_COUNT register is at address 0x0C000004. It increments after each instruction executes and is reset to zero once it reaches the TIMER\_LIMIT.

The TIMER\_LIMIT register is at address 0x0C000008. It is used to specify how often an interrupt should be generated. NOTE: Be sure the limit is high enough so that the interrupt service routine can finish before another interrupt is signaled.

# Interpreter Architecture

This section describes some of the implementation details of the interpreter. It does not define the behavior of the virtual machine, but rather how the virtual machine is implemented.

## Memory

In order to facilitate memory management, all memory references are made through the following four functions:

Get\_Word Returns a word-sized value. The address must be word aligned.

Set\_Word Sets a word-sized value. The address must be word aligned

Get\_Byte Returns a byte-sized value. There are no alignment requirements on the address.

Set\_Byte Sets a word-sized value. There are no alignment requirements on the address.

If the address specified to these functions is outside the range of main memory, then the address will be interpreted as the address of a memory mapped IO register.

There are two functions for managing the machine state. They get/set all the registers. These functions are Get\_Machine\_State and Set\_Machine\_State.

## Executable file format and the Loader

To be specified

## Adding IO devices to the interpreter

To be specified

# Instruction Reference

This section defines the opcodes for the stackl machine. All opcodes occupy a word.

For binary arithmetic operations, the left operand is always pushed onto the stack first.

Some opcodes take a parameter from the instruction stream (that is, they are two word instructions). These instructions are indicated by <param> in the opcode name where <param> is replaced by the name of the parameter. The parameter can be found at Memory[IP+1].

In the descriptions of the opcodes, memory offsets are given in word-sized increments. In other words, “IP+1” means the word following the IP, not the byte following the IP. Offsets relative to the IP assume the IP is pointing at the opcode.

In the descriptions of the effects of the opcodes, the operators have the meaning defined by the C language.

The numerical value of specific opcodes should not relied upon because they are subject to change in future versions of stackl.

## ADJSP <amount>

This opcode is used to adjust the stack pointer. This opcode has the following effect:

SP += <amount>

The <amount> can be a positive or negative number. The <amount> is in bytes, not words. If <amount> is not a multiple of the word size, the machine behavior will be undefined following this instruction.

## AND

This opcode performs a logical AND. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] && Memory[SP-1]

SP--

## CALL <addr>

This opcode is used for making function calls. This opcode has the following effect:

Memory[SP] = IP+1 // return address

Memory[SP+1] = FP

SP += 2

FP = SP

IP = <addr>

## CALLI

This opcode is used for making function calls. The address of the called function is taken off the stack. This opcode has the following effect:

temp = Memory[SP-1]

SP -= 1

Memory[SP] = IP+1 // return address

Memory[SP+1] = FP

SP += 2

FP = SP

IP = temp

## CLID

This opcode is used to clear the interrupt disable bit in the FLAG register. This opcode has the following effect:

FLAG &= ~FL\_INT\_DIS;

IP += 1;

Following this instruction, interrupts will be enabled (meaning that if an interrupt occurs, the interrupt service routine will be called).

## DIVIDE

This opcode performs a divide. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] / Memory[SP-1]

SP--

## DUP

This opcode duplicates the value at the top of the stack. This instruction has the following effect:

Memory[SP] = Memory[SP-1]

SP++

## EQ

This opcode performs an equivalence check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] == Memory[SP-1])

SP--

## GE

This opcode performs a greater than or equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] >= Memory[SP-1])

SP--

## GT

This opcode performs a greater than check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] > Memory[SP-1])

SP--

## HALT

This opcode will cause the processor to stop executing instructions. The simulator will exit.

## ILLEGAL

This opcode will cause an illegal instruction exception and the simulator will exit.

## INP

This opcode will initiate an IO operation. The value at the top of the stack must be the address of an io\_blk. See the section on IO processing for a description of the IO blocks and their function.

## JMPUSER <dest>

This opcode causes the processor state to switch to user mode and then jump to an address in the user modes address space. This instruction has the following effect:

FL\_USER\_MODE bit is set in FLAG register

IP = <dest>

## JUMPE <dest>

This is the conditional jump instruction. It will jump if the value on the top of the stack is equal to zero. It has the following effect:

SP--

val = Memory[SP]

if (val == 0)

IP = <dest>

else

IP += 2

## JUMP <dest>

This is the jump instruction. It has the following effect:

IP = <dest>

## LAND

This opcode performs a logical (bitwise) AND operation. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] & Memory[SP-1])

SP--

## LE

This opcode performs a less than or equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] <= Memory[SP-1])

SP--

## LOR

This opcode performs a logical (bitwise) OR operation. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] | Memory[SP-1])

SP--

## LT

This opcode performs a less than check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] < Memory[SP-1])

SP--

## LXOR

This opcode performs a logical (bitwise) XOR operation. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] ^ Memory[SP-1])

SP--

## MINUS

This opcode performs a subtraction. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] - Memory[SP-1]

SP--

## MOD

This opcode performs a modulus. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] % Memory[SP-1]

SP--

## NEG

This opcode negates the value on the top of the stack. This instruction has the following effect:

Memory[SP-1] = - Memory[SP-1]

## NE

This opcode performs a not equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] != Memory[SP-1])

SP--

## NOP

This opcode performs no operation.

## OR

This opcode performs a logical OR. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] || Memory[SP-1]

SP--

## OUTS

This opcode sends output to the console. The value at the top of the stack is assumed to be the address of a NULL terminated string. The string at that address will be sent to the console. This is a blocking operation. The CPU will not execute another instruction until after the IO is complete.

## PLUS

This opcode performs an addition. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] + Memory[SP-1]

SP--

## POPCVARIND

This opcode pops a value off the stack and stores it in a byte-sized variable. The offset to the variable is taken from the stack. Note that a word-sized value is popped, but the destination is a single byte. To use this opcode, the value should be pushed on the stack and then the offset. The opcode has the following effect:

Memory[Memory[SP-1]] = Memory[SP-2]

## POPCVAR <offset>

This opcode pops a value off the stack and stores it in a byte variable. The offset to the variable is taken from the stack. Note that a word-sized value is popped, but the destination is a single byte.

The opcode has the following effect:

SP--

Memory[FP+offset] = Memory[SP]

## POPREG <reg num>

This opcode pops the value from the top of the stack into the specified register. The registers are numbered as follows:

0 BP register

1 LP register

2 IP register

3 SP register

4 FP register

5 FLAG register

## POPVAR <offset>

This opcode pops a value off the stack and stores it in the word indicated by FP+<offset>. The opcode has the following effect:

SP--

Memory[FP+offset] = Memory[SP]

## POP

This opcode pops the stack. The value that is popped in not saved. This opcode has the following effect:

SP--

## PUSHCVARIND

This opcode pushes a byte onto the stack. The offset to the byte comes from the stack. A single byte is pushed onto the stack but the SP is updated by a word-sized amount. The value of the unused three bytes is unspecified. To use this opcode, push the address of the value to be pushed onto the stack. This opcode has the following effect:

Memory[SP-1] = Memory[ Memory[SP-1] ]

SP -= 2

## PUSHCVAR <offset>

This opcode pushes a byte onto the stack. The byte is located at FP+<offset>. To preserve stack alignment, the SP is increased by a word-size amount. The value of the unused bytes is unspecified.

This opcode has the following effect:

Memory[SP] = Memory[FP+<offset>]

SP++

## PUSHFP

This opcode pushes the frame pointer onto the stack. This opcode has the following effect:

Memory[SP] = FP

SP++

## PUSHREG <reg num>

This opcode pushes the value of the specified register onto the stack. The registers are numbered as follows:

0 BP register

1 LP register

2 IP register

3 SP register

4 FP register

5 FLAG register

## PUSHVAR <offset>

This opcode pushes a word onto the stack. The word is located at FP+<offset>. This opcode has the following effect:

Memory[SP] = Memory[FP+<offset>]

SP++

## PUSH <value>

This opcode pushes a value onto the stack. This opcode has the following effect:

Memory[SP] = <value>

SP++

## RETURNV

This opcode is used to return from a function when the return function returns a value. The value to be returned is assumed to be at the top of the stack at the beginning of the instruction, and it is left on the top of the stack at the end of the function (after execution returns to the caller).

This opcode has the following effect:

temp = Memory[SP-1]

SP = FP-1

IP = Memory[FP-2]

FP = Memory[FP-1]

Memory[SP-1] = temp

## RETURN

This opcode is used to return from a void function. This opcode has the following effect:

SP = FP-1

IP = Memory[FP-2]

FP = Memory[FP-1]

## RTI

This opcode is used to return from interrupt mode. Both the interrupt and systrap functions should use this opcode to return.

## SEID

This opcode is used to set the interrupt disable bit in the FLAG register. This opcode has the following effect:

FLAG |= FL\_INT\_DIS;

IP += 1;

Following this instruction, interrupts will be disabled.

## SETMODE <mode>

This opcode changes the FLAG register. The instruction has the following effect:

FLAG = <mode>

IP += 2

## SHIFTL

This opcode performs a bit shift left. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] << Memory[SP-1]

SP--

## SHIFTR

This opcode performs a bit shift right. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] >> Memory[SP-1]

SP--

## SWAP

This opcode swaps the two items at the top of the stack.

temp = Memory[SP-2]

Memory[SP-2] = Memory[SP-1]

Memory[SP-1] = temp

## TIMES

This opcode performs a multiply. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] \* Memory[SP-1]

SP--

## TRAPTOC

This opcode is reserved and should not be used.

## TRAP

This opcode performs a trap. It has the effect of calling the systrap() function. See the syscall section of this document for more details.

# stacklc grammar

The stacklc grammar is defined below:

program: global\_decls

block: open stmts close

| '{' '}'

open: '{'

close: '}'

decls: decls decl

| decl

decl: var\_decl ';'

| struct\_decl ';'

| UNSUPPORTED

| error ';'

var\_decl: TYPE\_ID IDENTIFIER

| CHAR '\*' IDENTIFIER

| CHAR IDENTIFIER arrayspec

struct\_decl: TYPEDEF STRUCT open decls close IDENTIFIER

global\_decls: global\_decls global\_decl

| global\_decl

global\_decl: func\_decl

| struct\_decl

| CONST TYPE\_ID IDENTIFIER '=' INT\_VAL ';'

| DEFINE IDENTIFIER INT\_VAL

| DEFINE IDENTIFIER '-' INT\_VAL

| PRAGMA ONCE

| PRAGMA INTERRUPT IDENTIFIER

| PRAGMA SYSTRAP IDENTIFIER

| PRAGMA STARTUP IDENTIFIER

| PRAGMA FEATURE IDENTIFIER

func\_decl: func\_header ';'

| func\_header '{' stmts '}'

| func\_header '{' '}'

func\_header: func\_prefix paramsspec ')'

| func\_prefix ')'

func\_prefix: TYPE\_ID IDENTIFIER '('

func\_prefix: CHAR '\*' IDENTIFIER '('

paramsspec: paramsspec',' paramspec

| paramspec

paramspec: var\_decl

arrayspec: '[' INT\_VAL ']'

stmts: stmts stmt

| stmt

stmt: decl

| ';'

| IF '(' ccomp ')' stmt ELSE stmt

| IF '(' ccomp ')' stmt

| FOR '(' assign ';' ccomp ';' assign ')' stmt

| WHILE '(' ccomp ')' stmt

| assign ';'

| func\_call ';'

| block

| RETURN expr ';'

| asm\_stmt ';'

assign: lval '=' expr

| lval '=' string\_lit

| lval '=' asm\_stmt

| lval PLUS\_EQ expr

| lval MINUS\_EQ expr

| lval TIMES\_EQ expr

| lval DIVIDE\_EQ expr

| lval INC

| INC lval

| lval DEC

| DEC lval

asm\_stmt : ASM '(' INT\_VAL ')'

| ASM '(' INT\_VAL ',' params ')'

func\_call: IDENTIFIER '(' params ')'

| IDENTIFIER '(' ')'

varref: varref '.' varpart

| varpart

varpart: IDENTIFIER arrayval

lval: varref

arrayval: arrayval '[' expr ']'

| /\* empty \*/

params: params',' param

| param

param: expr

| string\_lit

| '&' varref

ccomp: ccomp OR comp

| ccomp AND comp

| comp

comp: comp EQ expr

| comp NE expr

| comp GT expr

| comp LT expr

| comp LE expr

| comp GE expr

| expr

expr: expr '+' term

| expr '-' term

| term

term: term '\*' value

| term '/' value

| term '%' value

| value

value: fact

| '-' fact

fact: '(' ccomp ')'

| INT\_VAL

| varref

| func\_call

| SIZE\_OF '(' IDENTIFIER ')'

| SIZE\_OF '(' TYPE\_ID ')'

| SIZE\_OF '(' CHAR ')'

| SIZE\_OF IDENTIFIER

string\_lit: STRING\_LIT