Grammar Parsing (EBNF)

Example: PEMDAS
Parsing – in what order to evaluate terms?
How do you parse
x * y + z / 4
as a mathematical expression?

WRONG INTERP
((x*y)+z)/4

RIGHT INTERP:
(x*y)+(z/4)

Grammar helps define the way terms are connected and should be parsed.

ENBF Grammar has
(1) Start symbol
(2) Production rules

Through repeatedly expanding the start symbol through chains of production rules, we produce the entirety of word combinations in the grammar.
Take an 'expr', 'term', or other nonterminal symbol and replace it by the production on the right hand side of a rule. For example.

expr
expr "+" term
term "+" term
(term "*" atom) "+" term
(atom "*" atom) "+" term
(atom "*" atom) "+" (term "/" atom)
(atom "*" atom) "+" (atom "/" atom)
(x "*" y) "+" (z "/" 4)

expr := term | expr "+" term | expr "-" term
term := atom | term "*" atom | term "/" atom

The grammar expresses that division and multiplication are 'higher precedence' than addition and subtraction, therefore the grammar enforces the priority of operations.

The grammar does not allow the incorrect parsing ((x+y) * z) / 4 to be produced, as it is impossible to produce it from production rules expanding the start symbol.

First code example:
parent prints (“hello from parent %d”, getpid())
child prints goodbye and exits
parent waits for child

usleep(500000) – sleeps for 500000 microseconds = .5 seconds

It works. What if we sleep for 5 seconds? Then the parent is stuck waiting until the child is done. What if we want the parent to wait at most .75 seconds, if the child hasn’t exited by then?

```
double start_time = timestamp();
int status;
pid_t exited_pid;
while (timestamp() – start_time < 0.75)
   exited pid = waitpid(pi,&status,0);
assert(timestamp() – start_time >= .75 || exited_pid == p1);

if (timestamp() – start_time >= .75)
    fprintf(“stderr,”Child bored me.\n”);

if (WIFEXITED(status))
   fprintf(stderr,”Child exited with status “%d after %d sec\n”,WEXITSTATUS(status),timestamp() –
start_time);
```
---

timestamp() prints number of seconds since January 1, 1970 (UNIX epoch begins). Wait for the child for .75 seconds, then print out that the child bored the parent if the child didn’t exit.

It waits for the child to exit before printing out that the child bored me. That’s because the parent is still running wait_pid. We should use the flag WNOHANG to make sure wait_pid doesn’t block.

Blocking: System call does not return, and the process that ran the system call is left waiting.

System calls that cause blocking:
waitpid(), read() blocks when a pipe is empty or a file hasn’t been brought to the disk, atomic system calls, write can block, usleep blocks.

To see that the child process is blocking, we will use ps auxww | grep wait. We see that the S+ character means that the process is sleeping.

(compare to program which does an infinite loop: then the S+ is replaced by R+ which means the process is running.)

Now the parent isn't blocking, so the parent exits after .75 seconds.

BUT if the child usleeps for only half a second…we want the parent to print that the child exited, nto that the child bored it. However, the loop (while timestamp() – start_time < .75) doesn't exit if the child exits. So add condition to break out of the loop.

```
if (exited_pid > 0)
   break;
```

"top" shows us that the wait program is using 99.9 of the system's resources because it's repeatedly checking if its child has exited.

This is bad utilization of resources. This type of bad utilization is known as polling.

Polling is the opposite of blocking; blocking puts the calling process to sleep until a condition is met, polling returns immediately with whether the event/condition has happened or not. With WNOHANG, waitpid becomes a polling system call.

One solution: poll, then sleep. Poll, then sleep. The parent will sleep and check periodically, every 10 milliseconds or so.

Two problems with this solution. This process is not completely blocking, since it's waking up every 10 ms. It's also not as responsive as a polling solution, since the parent *isn't* constantly checking. This compromise loses out on some of the advantages of full blocking and full polling.

We want an analogous model to interrupts.
SIGNALS:
Unix process-level abstraction for interrupts.

Every signal has name and number. SIGKILL (9) destroys a process with no recourse.
SIGALRM is an alarm; several system calls set the alarm.

ITIMER_REAL decrements in real time, delivers SIGALRM signal when the timer expires.

So add to the code:

```
struct itimerval itimer;
itemerclear(&itimer.it_interval);
itimer.it_value.tv_sec = 0;
itimer.it_value.tv_usec = 750000;
r = setitimer(ITIMER_REAL,&itimer,NULL);
assert(r >= 0);
```

Running wait now kills the process and prints 'alarm clock' because we didn't handle the signal; we default to killing the process and having the shell printing a message.

Putting a signal handler in…now the kernel will redirect the signal action to be a function that we create. Now wait returns early. All we did was receive SIGALRM.

When a process receives a signal, the process moves on with its life after executing the signal handler. Whatever system call was running when the signal was received returns early (hence waitpid returns early, as we want!) Since the child received the interrupt, it exits and sets 'errno' the global variable equal to EINTR, which says it was interrupted by a signal.

Even better, SIGCHLD is a solution already implemented in UNIX which is sent when a child changes status! Now we get interrupted exactly when the child exits.

Better solution:
int r = mysignal(SIGCHLD,signal_handler);

// now our program handles the SIGCHLD signal with a function which doesn't ignore the signal

The parent process now does the following:
r = usleep(750000);
if (r == -1 && errno == EINTR)
  fprintf(stderr,"usleep interrupted by signal %g sec\n",timestamp() – start_time) // child has exited

Another problem:
What if the child exits right away? Perhaps before the signal handler is set up? It is not determined whether the parent or child runs first, so the parent may wait even if the child exits right away.

RACE CONDITION: Bug dependent on scheduling

Even if we move the signal handler installation before the forking, we still have this race condition. This is because the wrong usleep gets interrupted. We only want to sleep those 750000 seconds if the signal didn't already get sent. Set a global variable as a flag, and change the signal handler to update this global variable?

This code is not free of race conditions. A signal could be delivered between any two instructions.
You can't 'make an atomic thing' in a process because that would be equivalent to a process turning off interrupts.

Some system calls are 'safe' to call within a signal handler. (Malloc is not safe, printf is not safe – a complete list is available in the man pages)

Writing global variables is safe!

The current race condition happens between these two lines:
if (!has_signal_happened){
r = usleep(750000);

…

if the signal is delivered (i.e. the child process dies) after the check but before the usleep, we will wait for .75 seconds when we shouldn't.

Use a pipe! A process can talk to itself this way. Write and read are safe system calls within signal handlers (i.e., they are atomic!) So instead of using a global variable as a storage of state, use a pipe, both ends of which are in the parent process. Instead of blocking on wait or usleep, we block on READ. The signal handler is now a WRITE into the pipe.

We need two signals now: the handler for the sigchild writes a byte into the pipe. The handler for the alarm writes a byte into the pipe. Whichever happens first will be read first from the pipe!