Minishell: Un subset de bash

jaicastr y mparra-s, 42 Madrid

Julio 2025

ABSTRACT

Este documento presenta una visión técnica de la implementación de minishell, una shell escrita en C que replica el comportamiento básico de Bash, aunque con un toque personal y una filosofía más acercada a la seguridad. Se describe la estructura interna del entorno, la gestión de variables, y el sistema de interpretación de la entrada del usuario. El sistema se basa en contenedores tipados dinámicamente, inferencia de tokens y un modelo de ejecución adaptable a distintos contextos.

El diseño de esta shell está especialmente pensado para interactuar con el usuario como si fuese un compilador o DSL de shell-lang.

Contents

1 Uso

1.1 Modo interactivo

Al ejecutar el binario sin argumentos, la shell entra en modo interactivo:

- `./minishell`

En este modo, se muestra un prompt personalizado con el usuario y el nombre del sistema. El usuario puede introducir comandos como en Bash. El entorno inicial se hereda desde `envp`, y se almacena internamente como un vector de variables.

1.2 Modo oneliner

Si se proporciona la opción `-c`, la shell ejecuta el comando indicado y termina:

- `./minishell -c "echo foo"`

Este modo es útil para scripts y pruebas automatizadas. El comando se interpreta de forma idéntica a una línea en modo interactivo, incluyendo redirecciones, tuberías y expansión de variables.

1.3 Opción `--debug`

Activa mensajes de depuración que se imprimen en stderr. Estos incluyen:

- Tokens generados durante el análisis léxico.
- Variables de entorno cargadas.
- Heredocs y redirecciones detectadas.
- Resultados del preprocesador
- Árbol de ejecución y AST

Permite seguir el flujo interno de la shell sin afectar su comportamiento.

1.4 Opción `--help`

Imprime una lista de funcionalidades soportadas, flags disponibles y notas de implementación. El contenido se muestra con colores ANSI para facilitar la lectura desde terminales compatibles.

2 Setup y Estructuras de Datos

2.1 Setup

Lo primero que hay que entender de los programas es que estos son llamados desde una función llamada `main`, y este `main`, tiene la siguiente firma:

```
1 int main(int argc, char **argv, char **envp)
```

Esta firma la usaremos más adelante para invocar programas desde el shell. Por ahora sólo nos interesa saber que `char **envp` es un conjunto de cadenas de texto, con un nombre y valor, separados por un '=', de la siguiente manera:

```
NOMBRE=VALOR
NOMBRE2=VALOR2
...
```

Cada una de estas entradas en `char **envp`, serán denominadas `variables de entorno`. Estas variables serán heredadas por cualquier programa al que la shell invoque, a su vez, el usuario puede crearlas y destruirlas utilizando `export y unset`, respectivamente. También estarán expuestas al usuario como parte de la sintaxis de la shell, cuando una palabra tiene el prefijo `$`, se referirá a la variable con ese nombre, en esta implementación y en todas las implementaciones serias se puede inhibir esta característica del operador si ustamos '\' como prefijo).

2.2 t_var

Durante el proceso de inicialización de la shell, esta guardará los nombres y las variables en dos `t_string` separadas entre sí, pero que pertenecen a la misma estructura `t_var`:

```
typedef struct s_var
{
    t_string    name;
    t_string    value;
}   t_var;
```

Durante el `runtime` del programa interactuaremos con con ellas mediante un vector `t_vec`, el cual nos permite acceder a los punteros en memoria continua y dinámica, lo que hace que mejore muchísimo la localidad del caché sobre listas encadenadas, por ejemplo, no obstante, el algorítmo de búsqueda implementado tiene una complejidad de `O(n)`, aunque no esperamos que el entorno crezca lo suficiente para que sea una pérdida de rapidez significante.

2.3 t_vec

La estructura `t_vec` que almacenará los tokens tiene la siguiente forma:

```
typedef struct s_vec
{
    size_t  size;
    void    *data;
    size_t  alloc_size;
    size_t  sizeof_type;
}   t_vec;
```

Este `t_vec` será el `env` y el `tokenstream` de nuestra shell.

2.4 t_data

La estructura que guarda el estado de la shell, con variables de acceso rápido al `env` y otros datos necesarios en *scopes* diferentes será `t_data`:.

```
typedef struct s_data
{
    t_string    prompt;
    char        **envp;
    bool        debug;
    bool        oneliner;
    bool        phelp;
    bool        hdoc_terminate;
    bool        under_valgrind;
    bool        exit;
    bool        segfault;
    char        *invocation;
    t_string    oneliner_s;
    t_string    hostname;
    t_var       lastcommand_res;
    t_vec       tokv;
```

```
17      t_vec       env;
18      t_string    *path;
19      t_string    *username;
20      t_string    *pwd;
21  }   t_data;
```

Esta estructura se pasa por argumentos a todas o casi todas las funciones de la shell, para en todo momento ser capaces de saber en qué estado nos encontramos y como debemos proceder respecto a éste.

2.5 t_tok

Estas unidades léxicas están compuestas por `tokens`, que en este caso son un conjunto de un `enum t_toktype` y una `t_string`. El primero representa un tipo semántico inferido, usado para clasificar partes del input del usuario, y puede cambiar durante el análisis.

```
1  typedef struct s_tok
2  {
3      t_toktype   type;
4      t_string    s;
5  }   t_tok;
```

El enumerado que representa los tipos posibles de tokens es:

```
1  typedef enum e_toktype
2  {
3      TOK_IDENT,
4      TOK_STRING,
5      TOK_STRING_DQ,
6      TOK_STRING_SQ,
7      TOK_STRING_EMPTY,
8      TOK_DQ,
9      TOK_SQ,
10     TOK_LR,
11     TOK_RAPPEND,
12     TOK_REDIR,
13     TOK_HDOC,
14     TOK_RR,
15     TOK_AND,
16     TOK_AMPER,
17     TOK_PIPE,
18     TOK_EQ,
19     TOK_OR,
20     TOK_LPAREN,
21     TOK_RPAREN,
22     TOK_LCURLY,
23     TOK_RCURLY,
24     TOK_DOLLAR,
25     TOK_SCOLON,
26     TOK_VAR,
27     TOK_STRING_TOEXPAND,
28     TOK_REDIR_NN,
29     TOK_REDIR_IN,
30     TOK_WRITE_IN,
31     TOK_TILDE,
32     TOK_REDIR_TO,
33     TOK_APPEND_TO,
34     TOK_REDIR_FROM_FD,
35     TOK_APPEND_FROM_FD,
36     TOK_SUBSTITUTION,
37     TOK_SUBS_START,
38     TOK_SPACE,
```

```
39 }    t_toktype;
```

Todos estos tipos de token nos permiten saber de manera expresiva qué es lo que quiere decir el usuario, estos tokens viven en un `stream`, que se someterá a reglas, clasificaciones y modificaciones a lo largo del programa.

2.6 t_string

Las strings las representamos de tal manera que en todo momento sabemos cuánto miden, haciendo el acceso mucho mas seguro y rápido, ya que no tenemos que recalcular su longitud constantemente.

```
1 typedef struct s_string
2 {
3     size_t  len;
4     size_t  alloc_size;
5     char    *data;
6 }   t_string;
```

Además, este contenedor sigue los mismos principios que los del vector, lo que quiere decir que también es una estructura que crece y decrece de manera dinámica.

3 Fases del intérprete

La interpretación del input se divide en 4 fases, las cuales tienen un sólo propósito y están muy marcadas.

3.1 Análisis léxico (Lexer)

El lexer de `minishell` convierte una cadena de entrada en una secuencia tipada de tokens, con una arquitectura basada en funciones tipo "eater" que actúan como estados independientes de una máquina de análisis. Cada tipo de secuencia se identifica y procesa por separado, permitiendo un análisis robusto de comandos, comillas, operadores, variables y espacios.

3.1.1 Estrategia general

El flujo principal se ejecuta en la función `lex()`, que recorre la entrada carácter por carácter y delega a funciones auxiliares según el tipo de token detectado. El resultado es un `t_vec` de estructuras `t_tok`, cada una con su tipo y contenido.

3.1.2 Tipos de token

Cada token es identificado mediante la estructura ya descrita `t_tok`.
Los tipos posibles (`t_toktype`) incluyen identificadores, strings entrecomillados, operadores como `|` o `>>`, y tokens especiales como paréntesis, variables, o comentarios. Para depuración, se incluyen funciones como `get_token_pretty()` que mapean cada tipo a su representación en texto legible.

3.1.3 Máquina de estados modular

La función `lex()` usa un pipeline de funciones con prefijo `try_lexas_` que actúan en cascada:

- `try_lexas_spc` : reconoce espacios y los representa como tokens.
- `try_lexas_qs` : detecta strings con comillas simples o dobles, manejando escapes y errores de cierre.
- `try_lexas_ident` : consume identificadores y palabras sueltas, permitiendo caracteres extendidos.
- `try_lexas_op` : identifica operadores como `>>`, `&&`, `;` y similares.

- `try_lexas_comment` : descarta comentarios iniciados con `#`.

Cada función avanza un offset en la entrada y empuja, si corresponde, un nuevo token al vector de salida.

3.1.4 Sanitización y errores

Las funciones `eat_string_dq`, `eat_string_sq` y similares consumen delimitadores de strings y comprueban su cierre. En caso de error de sintaxis (como comillas sin cerrar), se devuelve `SIZE_MAX` y se limpia la secuencia de tokens con `clean_tokenstream()`.
Además, se realiza una comprobación de paréntesis con `manage_paren()`, que mantiene un contador y detecta si hay desbalanceo.

3.1.5 Compatibilidad extendida

Se permite una gran variedad de caracteres en identificadores, incluyendo `~`, `+`, y caracteres no ASCII, para robustez frente a entradas inesperadas. Los operadores soportan combinaciones como `>>`, `&&` o `>|`, mapeadas con precisión mediante funciones como `get_token_type_1` y `get_token_type_2`.

3.1.6 Depuración

Al activar la opción `--debug`, la shell muestra los tokens generados en tiempo real, ayudando a depurar errores en la entrada o entender cómo se ha interpretado una línea.

3.2 Expansión y Preprocesado

Una vez finalizado el análisis léxico, el vector de tokens resultante se somete a una fase de preprocesado que se encarga de transformar la secuencia en una forma más semánticamente clara, eliminando ambigüedades, colapsando secuencias y expandiendo variables. Esta fase actúa como un DSL transformer que convierte el texto "bash-like" en un formato más controlado.

3.2.1 Colapso de cadenas y espacios

El preprocesador comienza unificando tokens consecutivos de tipo string o identificador que no están separados por espacios. Esto permite convertir secuencias como:

```
echo "foo""bar"
```

en una única cadena `"foobar"`. Esto se realiza en `strings_concat()`.
Después, se eliminan espacios que no aportan significado, especialmente aquellos que rodean redirecciones o asignaciones. Esta limpieza se realiza cuidadosamente para no interferir con estructuras como `2 > archivo`.

3.2.2 Expansión de variables

Una vez limpiado, se detectan secuencias como `$VAR` o `"$VAR"`. Usando funciones como `detect_vars()` y `expand_vars()`. Se soporta expansión explícita con comillas dobles, y se manejan secuencias escapadas como `\$VAR` para evitar la expansión. Las variables se extraen del entorno interno (`t_vec env`) y se reemplazan por su valor correspondiente en el token stream.

En caso de no existir la variable, el token se reemplaza por una cadena vacía.

3.2.3 Sustitución de comandos y restricciones

Aunque la shell no soporta sustituciones de comandos como `$(...)`, se detectan estas secuencias y se eliminan con un mensaje de advertencia, preservando la integridad del tokenstream y evitando errores de parseo posteriores.
También se bloquean estructuras no soportadas como `if`, `for`, o `while`, y se lanza un error de sintaxis si aparecen.

3.2.4 Expansión de wildcards

Si un token contiene solo el caracter `*`, se sustituye por una lista de nombres de archivo en el directorio actual (omitidos los ocultos). Si no hay coincidencias, se muestra un error.

3.2.5 Asignaciones y export

El parser soporta expresiones del tipo `export foo=bar`, las cuales son interceptadas y eliminadas del token stream. Su efecto colateral es insertar el valor en el entorno como una variable persistente. Para evitar ambigüedades, se invalidan formas como `export foo= bar` o `export foo=bar otro`, ya que rompen el modelo semántico. Estas reglas están implementadas en `varexp_parser()`.
Una vez cargada la variable, se inserta un token especial con valor `_sh__builtin_export` que será ignorado durante la ejecución.

3.2.6 Validación de contexto

Antes de continuar con la ejecución, se validan múltiples restricciones sintácticas:

- Redirecciones deben ir seguidas de archivos o números.
- Heredocs no pueden empezar una línea ni seguir a operadores.
- Heredocs no pueden preceder a operadores.
- Operadores deben ir entre comandos, nunca al inicio o fin.
- El último token debe ser una cadena, redirección o paréntesis de cierre.
- Todos los archivos que se redirigen al input deben existir.
- Todos los binarios deben ser encontrados o bien con resolucion absoluta on en el `$PATH`

Estas validaciones se realizan en `extra_checks()` y `catch_forbidden()` y garantizan que el stream sea coherente antes de construir el árbol de ejecución.

3.2.7 Transformaciones finales

Al final del preprocesado, los tokens que representan comandos incorporados como `unset`, `cd`, `pwd`, etc. son reemplazados por versiones internas como `_sh__builtin_unset`, lo cual permite tratarlos como comandos normales pero identificarlos fácilmente en el árbol de ejecución.
También se resuelve el path de los ejecutables usando `PATH` del entorno, permitiendo ejecutar tanto binarios absolutos como comandos disponibles en el entorno del usuario.

3.3 Parser y Árbol de Ejecución (AST)

Una vez finalizada la fase de expansión, el stream de tokens se convierte en un árbol de ejecución (AST, Abstract Syntax Tree). Este árbol representa la jerarquía y estructura lógica de los comandos que el usuario ha introducido.

3.3.1 Objetivo del parser

El parser convierte una lista lineal de tokens en una estructura binaria jerárquica que permite representar tuberías y operadores lógicos (`&&`, `||`) de forma anidada. Esta estructura facilita la ejecución recursiva y el control de flujo.

3.3.2 Tipos de nodo

El AST está compuesto por nodos del tipo `t_node`, que puede representar:

- **NODE_CMD**: un comando con argumentos y posibles redirecciones.
- **NODE_OP**: un operador lógico (`&&`, `||` o `|`) que conecta dos expresiones.

Cada nodo tiene una unión `u`, con los siguientes campos según el tipo:

```
typedef struct s_node
{
    enum { NODE_CMD, NODE_OP } type;
    union {
        t_cmd    *cmd;
        t_opnode *op;
    } u;
}   t_node;
```

3.3.3 Construcción del árbol

El parser sigue una gramática simple:

- `expr ::= '(' expr op expr ')' | pipe_expr`
- `pipe_expr ::= cmd ('|' cmd)*`
- `cmd ::= IDENT+ [redir]*`

La entrada comienza en `parse()`, que llama a `parse_expr()`. Si el primer token es un paréntesis, se interpreta como una expresión lógica. Si no, se interpreta como una secuencia de comandos conectados por tuberías.

3.3.4 Operadores y paréntesis

Para evitar ambigüedad, todos los operadores lógicos deben estar envueltos en paréntesis. Por ejemplo:

```
((echo foo | cat -e && ls) || echo bar)
```

El parser lanzará un error si se intenta usar `&&` o `||` fuera de paréntesis. Esto permite un parseo más limpio y un árbol estrictamente binario.

3.3.5 Pipelines y comandos

Las tuberías se procesan en `parse_pipe_expr()`, generando nodos de tipo `NODE_OP` con el operador `TOK_PIPE` y subnodos a izquierda y derecha.
Los comandos en sí se procesan en `parse_cmd()`, que extrae argumentos consecutivos y los guarda en `t_cmd->argv`. Si el siguiente token es una redirección, se construye un vector de `t_tok` con los datos relevantes.

3.3.6 Redirecciones

Durante el parseo del comando, las redirecciones se detectan y almacenan en `redir_v`, un vector interno del nodo comando. Estas se extraen del token stream y no participan en el árbol lógico, sino que afectan a la ejecución.

3.3.7 Visualización del árbol

Con la opción `--debug` activada, el árbol generado se imprime en stderr con identación y colores. La función `print_tree()` recorre recursivamente los nodos e imprime comandos, argumentos y redirecciones.
Ejemplo de salida:

```
CMD: [echo, hola] REDIRS: [] ARGC: 2
|_
  OP: &&
  |_
    CMD: [ls] REDIRS: [] ARGC: 1
  |_
    CMD: [wc, -l] REDIRS: [] ARGC: 2
```

3.3.8 Limpieza y errores

En caso de error durante el parseo, se libera el árbol parcialmente construido con `free_tree()`. También se detectan errores como falta de paréntesis o comandos vacíos.

3.3.9 Restricciones

Algunas restricciones clave impuestas por el parser:

- El token final debe ser consumido por completo.
- Las expresiones lógicas deben ir entre paréntesis.
- No se permite un operador sin lado derecho o izquierdo.
- Los comandos deben tener al menos un argumento válido.

Estas reglas permiten evitar ambigüedades y errores de ejecución en tiempo de shell.

3.4 Ejecución del AST

Una vez construido el árbol de sintaxis abstracta (AST), se entra en la fase de ejecución. Esta etapa recorre el árbol y ejecuta cada nodo de forma recursiva, aplicando redirecciones, ejecutando comandos según el tipo de nodo.

3.4.1 Recorrido del árbol

La ejecución se inicia con `run()`, que recibe el nodo raíz, el estado global (`t_data`) y un descriptor de entrada opcional. El flujo de ejecución depende del tipo de nodo:

- **NODE_CMD**: se ejecuta un comando individual, ya sea externo o un builtin.
- **NODE_OP**: se evalúa una operación binaria como `&&`, `||` o `|`.

3.4.2 Operadores binarios

Los operadores se manejan recursivamente:

- `&&`: se ejecuta el lado izquierdo, y si tiene éxito, se ejecuta el derecho.
- `||`: se ejecuta el derecho sólo si falla el izquierdo.
- `|`: se conecta la salida del izquierdo con la entrada del derecho usando un pipe.

3.4.3 Pipelines

El operador `|` se ejecuta mediante una doble bifurcación:

1. Se crea una tubería (`pipe()`).
2. El hijo izquierdo redirige su `stdout` al extremo de escritura y ejecuta el comando.
3. El hijo derecho redirige su `stdin` al extremo de lectura y ejecuta el siguiente comando.

Esto se implementa en `run_pipeline()` y usa funciones auxiliares como `fork_left()` y `fork_right()`.

3.4.4 Redirecciones

Cada comando puede tener un vector de redirecciones. Se aplica justo antes de ejecutar el binario, usando la función `make_redirs()`. Se manejan los siguientes tipos:

- `>` `TOK_REDIR_TO` → redirige `stdout` a archivo.
- `>>` `TOK_APPEND_TO` → añade al final del archivo.
- `<` `TOK_REDIR_IN` → redirige `stdin` desde archivo.
- `<<` `TOK_WRITE_IN` → simula heredoc en memoria.
- `n>archivo` `TOK_REDIR_FROM_FD` → redirige el descriptor `n`.
- `n>>archivo` `TOK_APPEND_FROM_FD` → añade `n` al final del archivo.
- `n>&m` `TOK_REDIR_NN` → redirige el descriptor `n` a `m`.

Estas redirecciones pueden estar codificadas como `"archivo:n"` para indicar el descriptor explícito. Esta convención se usa internamente en el parser para simplificar el análisis.
Si un comando tiene varias redirecciones del mismo fd, la última explicitamente toma preferencia sobre todas las demás, es decir:

```
echo bar | cat < foo.txt
```

En este caso sólo se tendría en cuenta `foo.txt`, ya que `cat` es invocado en un hilo que nace con `stdin` redirigido, al hacer las redirecciones explícitas del comando, se sobreescribe el descriptor `stdin` del hilo.

3.4.5 Ejecución de comandos

La ejecución de un nodo comando se hace en `run_cmd()`:

- Se llama a `fork()` para crear un proceso hijo.
- En el hijo, se aplican redirecciones y se llama a `execve()`.
- El padre espera al hijo y guarda el código de salida en `data->lastcommand_res`.

Si el comando es un builtin, se detecta por el prefijo `_sh__builtin_` y se ejecuta directamente con `run_builtin()`.

3.4.6 Builtins especiales

Algunos comandos integrados como `cd` y `exit` no pueden ejecutarse en procesos hijos, ya que alteran el estado global. Por eso se manejan directamente en `run_builtin()` sin fork.
Los demás builtins como `echo`, `pwd` o `env` se ejecutan en procesos hijos usando `run_normal_builtin()`.

3.4.7 Estado de salida y limpieza

Tras cada ejecución, se guarda el código de retorno en el entorno global. También se limpian los descriptores abiertos y las estructuras en el hijo con `child_cleanup()` para evitar fugas.
La función `load_last_result()` convierte el código numérico a string y lo guarda como `$?` interno de la shell.

3.4.8 Resumen del flujo

1. `run()` recorre el árbol.
2. Ejecuta los nodos comando o operador.
3. Aplica redirecciones con `dup2()`.
4. Forkea si es necesario.
5. Ejecuta el comando con `execve()` o función builtin.
6. Espera al proceso, guarda resultado y libera memoria.

4 Ciclos de vida de la memoria

Toda la memoria en la ejecución tiene asignado un tiempo de vida, las más remarcables son:

4.0.1 `t_vec` tokv

Este es el vector en el que viven los tokens, se reusan los allocs lo máximo posible, a menos que el vector tenga que crecer, o que se hayan de inyectar tokens en medio, no se cambia la región de memoria, lo que hace que el programa corra casi siempre en lineas de cache L1/L3.

4.0.2 `t_data` data

Vive durante todo el programa, misma región del stack de main() para un acceso mucho más rápido y eficiente.

4.0.3 `t_string` hdoc_ret

Durante todo lo que vive un heredoc, vive la string que almacena los datos temporales antes de escribirlos al vector de tokens, se reusa el buffer.

5 Bucle principal y control de entrada

Esta sección engloba la lógica de control general de la shell, incluyendo la generación del prompt, lectura de comandos, ciclo principal de ejecución y el manejo de constructos como `heredoc`. Es el corazón interactivo del programa.

5.1 Bucle principal (`core_loop`)

La función `core_loop()` ejecuta el ciclo de lectura y evaluación de comandos. En cada iteración:

1. Se genera el prompt con `default_prompt()`.
2. Se limpia el vector de tokens anterior.
3. Se lee una línea con `read_l()`, que almacena directamente los tokens.
4. Si no hay tokens, se reinicia el ciclo.

5. Se aplica el preprocesado con `pre_process()`.
6. Se maneja el heredoc, si existe.
7. Se resuelven rutas con `resolve_path()`.
8. Finalmente, se ejecuta el árbol con `parse_and_run()`.

5.2 Comandos en línea

La función `handle_oneliner()` permite evaluar directamente una cadena como comando sin pasar por el ciclo completo. Se utiliza, por ejemplo, para el modo script. Aplica exactamente el mismo flujo que el bucle principal, pero con una cadena ya cargada.

5.3 Prompt interactivo

El prompt se compone dinámicamente con:

- `default_prompt()` para el prompt principal.
- `hdoc_prompt()` para entradas heredoc.

El prompt incluye:

- Nombre de usuario y host.
- Directorio actual.
- Código de salida anterior, codificado por color:
  - Magenta si `$? == 0`
  - Rojo si `$?` $\neq$ `0`.

5.4 Manejo de `heredoc`

El soporte de `<<` se implementa sin archivos temporales: el contenido se almacena en memoria y se redirige con `pipe()`.
El ciclo de ejecución de heredocs consta de:

1. `look4hdoc()` busca el primer `<<`.
2. `heredoc_routine()` valida la secuencia de cierre.
3. `hdoc_loop()` pide líneas al usuario hasta que se encuentra la terminación.
4. Cada línea se almacena en memoria y se inyecta como tokens.

6 Filosofía

La filosofía de esta minishell se aleja de una mentalidad puramente centrada en los estándares POSIX y se centra más en principios como los de `rust`, ya que sólo pasaremos a la siguiente fase si no hay ningún fallo de ningun tipo.

Es decir, evitamos ambiguedades y sólamente aceptamos input que sabemos que es válido.