Compiladores

Linguagem *CompSh*

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Abril de 2025

Apresentação

Com este trabalho pretende-se desenvolver uma linguagem de programação compilada – i.e. que crie programas equivalentes ao programa a compilar numa linguagem de programação genérica (Java, Python, ...) – para lidar com a execução e teste de programas externos. Podemos dizer que se pretende desenvolver uma espécie de linguagem *shell* compilada, que para além de estar especialmente bem adaptada para executar programas, suporte tipos de dados numéricos e de texto com as operações aritméticas habituais.

A execução de um programa envolve três canais *standard* de comunicação do programa com o exterior: um de entrada – *standard input* – e dois de saída – *standard output* e *standard error*. Para além desses canais de comunicação, um programa quando termina a sua execução devolve um valor inteiro para o sistema operativo, indicando uma execução bem sucedida[1] com o valor zero, ou uma falha de execução (com um valor diferente de zero).

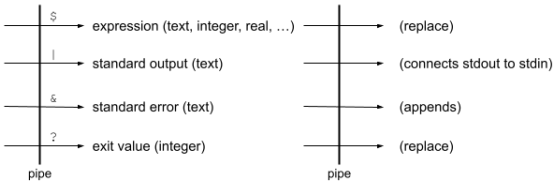
Entre outras alternativas, as linguagens shell suportam a execução combinada de programas recorrendo a um operador especial – *pipe* – que faz com que a saída *standard* de um programa passe a ser a entrada *standard* de outro (i.e. liga o *standard output* com o *standard input*). Em UNIX os *pipes* são uma forma de comunicação entre processos por mensagens. No entanto, neste trabalho, vamos fazer uma simplificação assumindo que o processo antes do pipe (se algum), termina antes dos valores sobre propagados pelo mesmo (i.e. toda a execução é sequencial).

```
ls | sort # bash example (not a CompSh program)
```

[1] Seja lá o que for que isso signifique para o programa.

Por outro lado, pretende-se que esta linguagem tenha suporte integral para operações com números e texto (à imagem das linguagens de programação habituais).

Assim, uma das generalizações desta linguagem consiste em permitir valores contendo quatro canais (em paralelo): uma expressão [$] (que pode ser do tipo texto, inteiro, real, e, eventualmente, outros), o *standard output* [|] (texto), o *standard error* [&] (também texto), e um valor inteiro com o resultado da execução do programa [?] (*exit value*). Para manter a coesão forte com o objectivo da linguagem, considera-se também que, existindo o canal *exit value*, então a sua utilização é também equivalente a uma expressão booleana (verdadeira se o valor for zero e falsa no caso contrário). Dessa forma, pode-se usar a execução de um programa directamente numa (eventual) instrução condicional significando uma execução bem sucedida do mesmo.



Esta linguagem pretende também generalizar o operador *pipe* de múltiplas formas:

- permitir o encaminhamento de diferentes combinações dos quatro canais de comunicação (em vez de apenas um, em geral o *standard output*), possibilitando a aplicação da operação seguinte apenas a um dos quatro canais (mas, por omissão, deixando passar os restantes para a operação seguinte).

- implementar as operações de escrita, leitura e atribuição de valores a variáveis nos *pipes*:

```
"Hello World!" | stdout          # write in standard output
"Hello World!" | stderr          # write in standard error
stdin "Name: " | stdout          # write text from stdin in stdout
name: text                       # declare variable name of type text
stdin "Name: " | store in name   # store text in variable name
```

- permitir activar para a operação seguinte apenas um dos quatro canais possíveis (a validação semântica garante a sua existência):

```
res: program          # declare variable res of type program (four possible channels)
!"ls"! | store res    # execute external program ls and store the result in res
res || stdout         # write stdout channel of res in stdout
res |& stdout         # write stderr channel of res in stdout
res |? stdout         # write exit value channel of res in stdout
1+2*3 |$ stdout       # write expression (integer type) channel of res in stdout
1+2*3 | stdout        # same as previous instruction (only exists the expression channel)
!"ls"!                # execute external program, without *any* effect in csh
```

Por exemplo, podemos definir o seguinte programa:

```
!"ls"! | stdout          # write ls results in stdout
prog: text;              # declares variable prog of text type. ; is optative
"find" | prog            # assigns text to variable prog
!prog! | store res: program  # store result of program in variable res
res || stdout            # only send program's stdout text to stdout (not a minimum requirement!)
res |? stdout            # only send program's exit value to stdout (not a minimum requirement!)
```

A descoberta da sintaxe desta linguagem deve ser feita recorrendo os programas de exemplo.

A linguagem secundária (interpretada) – `ISh` – vai ser uma versão interpretada da linguagem principal.

```
# min01.ish
integer(stdin "a: ") | store in a: integer;
integer(stdin "b: ") | store in b: integer;
a+b | stdout
```

A linguagem `CompSh` vai ter uma instrução para executar programas `ish`:

```
!!"min01.ish"!! | stdout # execute a ish program
```

Chama-se a atenção de que eventuais variáveis em programas `ish` não devem ter nenhum efeito no código `csh` que o executa (e vice-versa).

Características da solução

Apresentam-se a seguir um conjunto de características que a solução desenvolvida pode ou deve contemplar. Essas características estão classificadas a 3 níveis:

- mínima – característica que a solução tem obrigatoriamente que implementar;
- desejável – característica não obrigatória, mas fortemente desejável que seja implementada pela solução (apenas considerada se as mínimas forem cumpridas);
- avançada – característica adicional apenas considerada para avaliação se as obrigatórias e as desejáveis tiverem sido contempladas na solução.

Características mínimas

Os exemplos que começam por `min` (`min011.csh`, `min02.csh`, `min01.ish`, etc.) indicam algum código fonte que tem de ser aceite (e devidamente compilado e interpretado) pelas linguagens a desenvolver.

A linguagem deve implementar:

- Instrução para executar um programa externo: `!"ls"!`.

- Os tipo de de dados texto (`text`), inteiro (`integer`), real (`real`) e programa (`program`).
- Aceitar expressões aritméticas standard (e parêntesis) para os tipos de dados numéricos. Deve também aceitar a operação de concatenação de texto com o operador `+`.
- Operador *pipe* aplicável (pelo menos) uma vez e sem discriminar o canal activo. Este operador serve de base para as operações de escrita, leitura e atribuição de valor a variável.
- Instrução de escrita no *standard output* e outra para o *standard error*.
- Instrução de leitura de texto a partir do *standard input*.
- Operadores de conversão entre tipos de dados (por exemplo, `text(10)` para converter para texto; ou `integer("10")` para converter para inteiro).
- Verificação semântica do sistema de tipos. Note que no caso das operações envolvidas num *pipe*, apenas as expressões que tenham activo um canal de entrada (stdin) é que podem aparecer depois de um pipe (é o caso do `stdout`, `stderr`, atribuição de valor a variável, etc.)

Para ajudar na descoberta de como se podem executar programas externos em Java, são fornecidos pequenos exemplos[2].

A linguagem interpretada deve aceitar programas similares ao da linguagem compilada. Para não complicar o processo, considere que os programas na linguagem interpretada, tal como na versão compilada, têm sempre origem em ficheiros (i.e. não são executados instrução a instrução). Esta opção liberta o *stdin* apenas para a instrução de leitura da linguagem.

Características desejáveis

Os exemplos que começam por `des` (`des01.csh`, etc.) indicam algum código fonte que se enquadra nas características desejáveis.

- Permitir a definição de expressões booleanas (predicados) contendo, pelo menos relações de ordem e operadores booleanos (conjunção, disjunção, etc.).
- Incluir a instrução condicional (operando sobre expressões booleanas).
- Incluir instrução iterativa tipo `loop` (operando sobre expressões booleanas).
- Generalizar a aplicação do operador *pipe* permitindo que se aplique as vezes que se quiser.

[2] ChatMOS.

```
!"ls"! | !"sort"! | store in res: program | stdout
```

- Implementar a selecção de canais a aplicar na operação após um *pipe*:

```
!"ls"! | store in res: program
res || stdout # send ls stdout to stdout
res |& stdout # send ls stderr to stdout
res |? stdout # send ls exit value to stdout
```

- Implementar o tipo de dados lista (`list`) e o valor literal:

```
# write list of java and antlr4 source code in current directory;
!"ls"!["*.java","*.g4"] || stdout
stdin "path: " | store in path: text;
!"ls"![path] || stdout
```

- Implementar filtros tipo `grep`, `sed` e operadores para inserir prefixos ou sufixos de texto:

```
# grep java in the stdout result of program ls:
!"ls"! || / "java" | stdout
# send ls stdout to stdout with 3sp indentation:
res || prefix "   " | stdout
# send ls stderr to stdout with --> indentation:
res |& prefix "-->" | stdout
# send ls exit value to stdout (with NL at the end)
res |? suffix NL | stdout
```

- Implementar a redireção de canais num *pipe*:

```
# redirect stderr to stdout in the result of ls:
!"ls"! &^| | store in res: program
# redirect expression channel to exit channel (and store in res variable):
3*(5-5) $^? | store in res: program
```

  A redireção consiste numa expressão `<src>^<dst>` em que `<src>` ou `<dst>` são um do 6 possíveis símbolos (desde que faça semanticamente sentido!): `$|&?*-`. O significado dos quatro primeiros símbolos já foi indicado, sendo que os símbolos `*` e `-` significam, respectivamente, todos e nenhum canal. Assim eliminar o canal *exit* será:

```
!"ls"! ?^- | stdout  # redirect exit channel to none
```

Características avançadas

- Implementar funções e variáveis locais às mesmas.
- Implementar uma tabela de símbolos que resolva o problema dos contextos de declaração.
- Lidar com erros (por exemplo, falha de execução de um programa por não existir, ou não ser um ficheiro executável). (Possibilidade a explorar: verificar se faz sentido usar o canal *stderr* para esse efeito.)

- Tornar a análise semântica aplicável aos pipes absolutamente completa. Isto é, usando o sistema de tipos, garantir (estaticamente) que as operações com pipes fazem sempre sentido (só se aplicam a canais activos e que estejam definidos, etc.).
- Aceitar execução de programas em bash, python, e eventualmente outras linguagens interpretadas:

```
!bash!"myprog.sh"!! | stdout
```

- ...