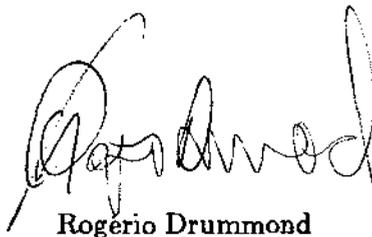


Compilação de Linguagens de Comando de Alto Nível

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Luiz Carlos Durso Carneiro e aprovada pela comissão julgadora.

Campinas, 07 de Abril de 1989



Rogério Drummond

Dissertação apresentada como requisito parcial para obtenção do Título de Mestre em Ciência da Computação pelo Instituto de Matemática Estatística e Ciência da Computação da Universidade Estadual de Campinas.

C215c

11378/BC

UNICAMP
BIBLIOTECA CENTRAL

COMPILAÇÃO
DE
LINGUAGENS
DE
COMANDO
DE
ALTO NÍVEL

Luiz Carlos Durso Carneiro
Orientador : Rogério Drummond

Março 1989

DEDICATÓRIA

Dedico este trabalho a meus pais : Inácio e Dalva
a meus irmãos : Cacau, Betinho e Riquinho
e a meus sobrinhos : Patrícia, Luiz Fernando, Carla, Carlos e
Marcelo.

AGRADECIMENTOS

Gostaria de agradecer a todos que direta ou indiretamente contribuíram para a execução deste trabalho, mas sei que corro o risco de esquecer alguém. Entretanto, existem algumas pessoas que contribuíram de uma forma toda especial e eu não poderia deixar de externar aqui o meu agradecimento.

A vocês, Rosilene, Paulinho e Rose, Eduardo e Beth, Marcos Euzébio, Osvaldo, Rogério (Zig), Zieghard, Alberto, Vicente, Glacir e Marcelo o meu muito obrigado pela convivência agradável durante estes anos. Vocês me proporcionaram ótimos momentos de descontração, contribuindo decisivamente para um bom andamento do meu trabalho.

Aos meus amigos do mestrado Fernando, Sérgio e Willians o meu muito obrigado pelo apoio e pelas cobranças :“ prá quando é a defesa?”.

Aos amigos do Grupo de Desenvolvimento de Ferramentas Furuti, Prof. João Meidanis e Prof. Rogério Drummond, o meu muito obrigado pelo apoio nas horas de dificuldade.

Gostaria de agradecer aos professores José Luiz Braga e José Geraldo, do departamento de matemática da UFV, pelo incentivo a pos-graduação e aos professores do departamento de ciência da computação da UNICAMP pela oportunidade de realizar este trabalho.

Finalmente gostaria de agradecer aos amigos do grupo CAPRICÓRNIO, Elza, Fábio, Luiz Otávio, Lúcia, Milton, Renata, Ronaldo, Sueli e Toninho, à Cris e ao Levi pelo apoio durante o ano em que trabalhei na Elebra Telecon.

Conteúdo

1	Introdução	2
1.1	Histórico do Desenvolvimento das Linguagens de Comando	2
1.2	Importância das Linguagens de Comando	4
1.3	Linguagens de Comando com Características de Linguagem de Programação	6
2	Descrição da C-SHELL : A Linguagem Escolhida como Protótipo	8
2.1	Introdução	8
2.2	Estrutura de Comandos	9
2.2.1	Comandos Simples	10
2.2.2	Redirecionamentos	10
2.2.3	<i>Pipeline</i> de Comandos	11
2.2.4	Comandos no <i>Background</i>	11
2.2.5	Execução Condicional	12
2.3	Substituições na Linha de Comando	12
2.3.1	<i>History</i>	12
2.3.2	<i>Alias</i>	13
2.3.3	Substituição de Variáveis	14
2.3.4	Substituição de Comando	14

2.3.5	Expansão de Nome de Arquivo	15
2.4	Expressões e Variáveis	15
2.5	Estruturas de Controle	18
2.5.1	Controle do Fluxo	18
2.5.2	Comandos Repetitivos	20
2.6	Entrada e Saída de Dados	21
3	Implementação	22
3.1	Implementação do Compilador	22
3.1.1	Introdução	22
3.1.2	Análise Léxica	23
3.1.3	Análise Sintática e Recuperação de Erro	24
3.2	Implementação da Biblioteca do Compilador	27
3.2.1	Implementação da Parte Dependente do Sistema Operacional	27
3.2.2	Implementação da Parte Independente do Sistema Operacional	37
4	Conclusão	46
4.1	Introdução	46
4.2	<i>Benchmarks</i> - Análise de resultados obtidos	48
4.2.1	Descrição Sumária dos Testes Efetuados	49
4.2.2	Análise dos Resultados	49
4.3	Extensões	50
A	Exemplos de Código Gerado Pelo Compilador	52
B	Descrição Sucinta dos Módulos do CCSH	56
B.1	Introdução	56
B.2	Descrição dos Módulos que Compõem o Compilador	57

B.2.1	Compilador	57
B.2.2	Biblioteca	57
B.3	Instalação do Compilador e da Biblioteca	62
B.3.1	Construção do Compilador	63
B.3.2	Construção da Biblioteca	64
C	Manual do Compilador CSH	65

Lista de Figuras

3.1	Esquema Geral do Processo de Compilação	23
3.2	Esquema da Integração	26
3.3	Esquema de Implementação de Redirecionamento de Entrada Padrão	31
3.4	Esquema de Implementação de Redirecionamento de Saída Padrão	31
3.5	Esquema da Implementação de <i>Pipeline</i> de Comandos	32
3.6	Esquema da Implementação de Comandos no <i>Background</i>	33
3.7	Esquema da Implementação de Substituição de Comandos	34
3.8	Estrutura de Dados para a Tabela de Símbolos	37
3.9	Estrutura de Dados para a Implementação de Vetores	38
3.10	Estrutura de Dados das Células	39
3.11	Estrutura de Dados para a Implementação da Pilha de Execução	40

resumo

É objetivo deste trabalho a apresentação do estudo da viabilidade de implementação de compiladores para linguagens de comando de alto nível.

As linguagens de comando têm se mostrado poderosas ferramentas de desenvolvimento nos ambientes de programação modernos, deixando para trás o conceito antigo de que serviam apenas para alocação de periféricos e execução de programas. Este estudo foi iniciado com o intuito de aumentar o potencial destas linguagens, tornando-as mais rápidas.

Este trabalho está organizado em 4 capítulos, onde são expostos vários aspectos do estudo das linguagens de comando e da implementação de um compilador protótipo.

No capítulo 1 é feita uma introdução às linguagens de comando, abordando tópicos que vão desde a sua origem até as modernas linguagens de comando de alto nível.

O capítulo 2 se constitui da apresentação da linguagem de comando escolhida como protótipo para implementação: a C-SHELL do sistema UNIX.

Já o capítulo 3 contém informações mais específicas sobre a implementação. Serão mostradas de modo distinto as partes da C-SHELL que foram implementadas utilizando recursos presentes no sistema UNIX e as partes independentes de sistema operacional. Estas discussões serão apresentadas a nível de algoritmos utilizados e quando for necessário, por questões de clareza, serão apresentados detalhes do código gerado.

Finalizando, o capítulo 4 é formado pela conclusão deste trabalho. Neste capítulo são discutidas as vantagens da compilação das linguagens de comando de alto nível. Serão apresentadas algumas comparações com as linguagens de comando disponíveis no nosso ambiente de programação UNIX e os novos passos a serem tomados nesta linha de estudo.

Nos apêndices poderão ser encontradas demonstrações de uso da linguagem, detalhes da composição dos diversos módulos do sistema e orientações para a instalação do compilador em outros sistemas UNIX e UNIX-LIKE.

Capítulo 1

Introdução

1.1 Histórico do Desenvolvimento das Linguagens de Comando

Praticamente todo sistema operacional é provido de uma linguagem de comandos através da qual seus usuários especificam as tarefas a serem executadas.

As linguagens de comando (LCs) variam de um sistema para outro, podendo ser bem simples como é caso do *CCP* (*Command Control Program*) do *CP/M* [DGT83] ou até mesmo bastante completas como é o caso das LCs do sistema *UNIX* [RIT78][THO78].

Nos primeiros sistemas operacionais as linguagens de comando eram bem rudimentares, tratando apenas da execução de programas e da alocação de periféricos. Seus comandos eram simples em relação à sua função, mas possuíam uma sintaxe extremamente complexa e ininteligível. A tarefa de corrigir os comandos de controle chegava a ser tão complexa quanto a tarefa de elaboração dos programas aplicativos.

A evolução dos sistemas operacionais provocou a necessidade da existência de linguagens de comando mais simples, flexíveis e com mais recursos. Como exemplo clássico desta necessidade pode-se citar as tarefas habituais que deveriam ser tomadas após a falha de um determinado programa: liberar dispositivos periféricos alocados, imprimir *dumps* de memória para análise posterior, etc. Atividades estas que não eram de simples execução

nas linguagens de comando da época. A necessidade de uma interface mais amigável fez com que novas facilidades fossem incorporadas às LCs.

Com a evolução dos sistemas operacionais e com a adição de novos recursos às LCs, várias tomaram a forma das linguagens de programação convencionais. Com isto apareceram linguagens de comando de alto nível com sintaxe simples e objetiva. São desta época as linguagens GEORGE 3 do ICL 1900 [BAR72] e a WORK FLOW LANGUAGE [COW75] (WFL) do Burroughs 6700/7700 que possuía uma sintaxe *Algol-like*. A WFL tinha uma outra particularidade interessante: ela foi uma das primeiras linguagens de comando a possuir variáveis, controle de fluxo, funções e opção para compilação.

A evolução das linguagens de comando têm tornado a interface com o usuário cada vez mais simples e amigável. Algumas chegam a oferecer um interfaceamento totalmente gráfico, como é o caso dos sistemas do Macintosh da Apple Computers, Star e XDE da Xerox e a WSH [BRE84] de ambientes UNIX. Nestes sistemas um simples acionar de um botão pode disparar a realização de qualquer tarefa. Outro tipo de interface encontrada em vários ambientes de programação e que é a preferida dos usuários não especialistas é a *menu shell*, onde o usuário escolhe a tarefa a ser executada dentre uma lista de tarefas possíveis.

É objeto de estudo recente [DRU87b] um novo modelo de interfaceamento do usuário com o sistema. Neste novo modelo, que é baseado no conceito de programação orientada por objetos, o usuário tem à sua disposição um conjunto de objetos básicos com os quais poderão ser compostos objetos mais complexos, como em um jogo de montar. Estes objetos básicos podem ser arquivos, programas, dispositivos periféricos e peças para conexão entre eles. Um aspecto interessante desta idéia é a facilidade com que novos programas podem ser montados graficamente a partir de programas mais simples, conectados de modo adequado através dos elementos de ligação. Um exemplo típico de elemento de ligação é o *pipe*, pois ele cria um canal de comunicação entre dois programas (objetos abstratos de alto nível). Este modelo de interfaceamento está sendo denominado de *Lego Shell* pela sua semelhança com o jogo de mesmo nome.

1.2 Importância das Linguagens de Comando

Uma parcela relativamente grande do uso de sistemas de programação é gasta interagindo com o interpretador de comandos, ou *shell*¹. A *shell* é, na maior parte do tempo, o único contato do usuário com o sistema operacional, sendo este julgado como um todo pelas facilidades e recursos oferecidos por esta linguagem. É necessário, então, que a LC explore ao máximo as potencialidades fornecidas pelo sistema operacional.

As LCs são de um modo geral as interfaces de mais alto nível de um sistema operacional. Elas devem possibilitar uma fácil utilização das abstrações criadas pelo sistema. Dependendo das suas potencialidades, uma LC deve ser capaz de manipular objetos abstratos, tais como arquivos, programas, dispositivos de E/S, processos, usuários e outros computadores.

Estas abstrações escondem os detalhes de sua implementação e apresentam uma interface simples com operações bem definidas. Em LC um programa pode ser visto como uma função que opera sobre seus argumentos - arquivos e/ou dispositivos periféricos (i.e., variáveis de tipos abstratos).

Programas complexos podem ser criados a partir de programas mais simples executados seqüencialmente se comunicando através de arquivos, fornecendo assim um ambiente propício à reutilização de programas.

Com a facilidade de manipulação de processos é possível especificar uma computação (ou parte desta) como um conjunto de tarefas a serem executadas concorrentemente, permitindo um aumento da eficiência através de um melhor aproveitamento dos recursos do hardware.

Se a LC suporta estruturas de controle e variáveis, então ela adquire as características de uma linguagem de programação convencional com a extensão dos tipos abstratos mencionados anteriormente. Estas extensões permitem criar um conjunto teoricamente ilimitado de comandos a partir dos programas existentes no sistema. Dado ao seu alto nível, certas aplicações podem ser especificadas e programadas em LC em um tempo inferior ao gasto com as linguagens de programação convencionais.

Obviamente todos os “tipos” aos quais nos referimos estão disponíveis no sistema operacional, mesmo que só potencialmente. É possível expressar qualquer programa em LC através de programas (em geral escritos em linguagem de montagem) que chamem as primitivas do sistema operacional.

¹Denominação geral das LCs de ambientes UNIX

Observe que o contrário não é necessariamente verdadeiro uma vez que a LC somente permite o acesso a certos objetos através de operações de alto nível, já que mantém estes objetos como tipos altamente abstratos. Esta opção, no entanto, exige um entendimento dos detalhes de baixo nível do sistema e, portanto, proibitiva para o usuário não especialista.

O problema é que apesar de todas estas possíveis potencialidades, poucos sistemas têm LCs suficientemente desenvolvidas que valham a pena serem utilizadas para programação. Por outro lado, como as LCs são interativas elas são interpretadas e, mesmo com o aparecimento de linguagens bastante expressivas e abrangentes, não existe, em geral, opção para compilação. Com a ineficiência inerente à interpretação uma série de programas apresentariam um custo proibitivo em tempo de execução.

Para ser adequada a LC não deve exigir declarações de variáveis e deve possibilitar que essas tenham "tipo" dinâmico. Como resultado prático destas "liberdades" o tempo de desenvolvimento em LC é menor que o tempo de desenvolvimento nas linguagens de programação convencionais (LPs). Assim a LC (como acontece em outras linguagens de altíssimo nível de abstração e relaxadas na necessidade de declaração e de preparação para a execução) torna-se um excelente instrumento de prototipagem de programas.

Se a LC for completa ela permitirá a construção de protótipos de programas elaborados. A existência de compiladores para estas linguagens possibilitará um aumento da eficiência dos protótipos e em muitos casos (programas pequenos) o produto final acabará sendo o próprio protótipo.

O que é realmente importante é que o processo de desenvolvimento tem seu tempo encurtado.

Estes foram alguns dos motivos que nos levaram a seguir esta linha de pesquisa. Escolhemos a C-SHELL do *UNIX* por ser bastante abrangente, servindo assim como protótipo para testes dos problemas existentes na compilação de LCs, e escolhemos o sistema *UNIX* por ser um ótimo ambiente de desenvolvimento de *software*.

1.3 Linguagens de Comando com Características de Linguagem de Programação

Na bibliografia sobre linguagens de comando é comum se encontrar duas correntes na discussão da viabilidade da existência de LCs com sintaxe semelhante às encontradas nas LPs. Alguns autores defendem esta unificação e outros são contrários. Serão mostrados a seguir os principais pontos desta discussão.

LPs e LCs tradicionalmente são distintas nos seus fundamentos mais básicos e nas suas áreas de aplicação. LCs são utilizadas para controle do sistema enquanto as LPs são utilizadas para desenvolvimento de aplicativos.

Em um nível mais baixo, LPs e LCs diferem mais radicalmente nas abstrações de dados que devem suportar. Tipicamente LPs com sintaxe *Algol-Like* suportam os tipos de dados fornecidos pela maioria dos computadores, tais como inteiro, real, caracter e vetores. As LCs tendem a suportar tipos de dados com um grau de abstração maior, tais como *strings*, arquivos, programas e processos.

Apesar destas diferenças existe uma série de fatos que tornaram a unificação desejável, tais como :

- A unificação das LCs com as LPs tornaria a aprendizagem de múltiplas linguagens desnecessária, pois removeria as distinções existentes entre elas e simplificaria a implementação em ambientes de programação.
- A execução de programas escritos em LC é imediata e fácil; após inserido o arquivo com os comandos ele está pronto para a execução. Uma vez criado um procedimento ele pode ser usado na construção de outros do mesmo modo que se usa os comandos internos da LC.
- Devido ao seu alto nível, procedimentos em LC são facilmente criados, testados e modificados.
- Os procedimentos criados em LC são utilizados do mesmo modo que os comandos internos.

Por outro lado as LCs na sua forma corrente não são suficientemente poderosas para muitos programas complexos e sofisticados. Existem áreas

em que elas são completamente deficientes, pois arquivos de comando (*shell scripts*) são executados lentamente devido ao fato de serem interpretados.

Apesar destes problemas algumas implementações, tais como a Lisp Shell [[ELL80] e [LEV80]], a linguagem EZ [FRA83], bem como as LCs encontradas no *UNIX*, são utilizadas com bastante frequência no desenvolvimento de aplicativos de controle e gerenciamento de sistemas.

Alguns autores ² acham que a principal vantagem de se ter LCs com sintaxe diferente das LPs se deve ao fato de que elas poderiam ser bem mais claras, facilitando o aprendizado destas linguagens pelos usuários não especialistas. Na maior parte do tempo eles utilizam estas linguagens para pequenos programas.

Outro fato que levou estes autores a criticarem a validade da existência de linguagens de comando de alto nível foi o de que a maioria dos usuários iriam preferir que a LC do sistema estivesse próxima de sua linguagem de programação predileta. Uma *shell* baseada na linguagem Lisp seria bem recebida por aficionados por esta linguagem, mas causaria pavor e pânico em grande parte dos usuários do sistema.

Esta crítica, entretanto, deixa de ser pertinente em sistemas como o *UNIX* em que a cada usuário é alocado um processo interpretador de linguagem de comando. Neste caso é possível oferecer aos usuários não especialistas uma *shell* restrita com interface através de *menus* e aos especialistas uma *shell* como a C-SHELL [JOY80] ou a BOURNE SHELL [BOU78].

²ver [MAL81]

Capítulo 2

Descrição da C-SHELL : A Linguagem Escolhida como Protótipo

Este capítulo faz uma breve descrição das principais construções presentes na C-SHELL. Para uma descrição mais detalhada das demais construções ver [SOB85] e/ou [JOY80]. Das construções apresentadas nessa seção somente o mecanismo de *history* não foi implementado na versão atual do compilador.

2.1 Introdução

Shell em *UNIX* é a denominação geral dos programas que promovem o interfaceamento dos usuários com o sistema operacional. C-SHELL é o nome de um dos interpretadores existentes neste sistema.

O objetivo principal de uma *shell* é a transformação de linhas de comando digitadas, ou lidas de um arquivo, em ações do sistema, tais como a execução de programas, alocação de periféricos, etc.

Dentre as LCs do *UNIX* a C-SHELL se destaca por ter sido a primeira a incorporar o mecanismo de *history*, facilidades de controle de tarefas e uma sintaxe semelhante à da linguagem C (vem daí o nome C-SHELL). Além destes recursos a linguagem permite diversas operações sobre arquivos, manipulação de periféricos, bem como entende do conceito de usuários e de

processos.

Cada processo está associado a um usuário e a alguns arquivos de E/S que podem ser substituídos por qualquer periférico pertinente. Processos podem ser criados, ter sua execução suspensa e terminados a qualquer tempo. Programas podem ser “concatenados” de forma que os dados de entrada de um programa sejam supridos pela saída de outros programas (*pipeline* de comandos).

É impossível falar em linguagem de comando no sistema *UNIX* e deixar de mencionar o nome da *BOURNE SHELL*, que foi a primeira *LC* do sistema *UNIX* a possuir uma sintaxe próxima à das linguagens de alto nível. Por isto e por ter sido lançada junto com as primeiras versões comerciais do sistema, ela foi, e provavelmente ainda é, a linguagem de comando mais difundida entre os usuários deste sistema. A *C-SHELL* surgiu alguns anos mais tarde na Universidade de Berkeley, e acrescenta à *BOURNE SHELL* uma sintaxe mais regular. Além da sintaxe, a *C-SHELL* inovou em relação à *sh* (denominação popular da *BOURNE SHELL*) no tratamento de variáveis simples e vetores, expressões aritméticas, no mecanismo de *alias* (sinônimos para comandos) e *history* (memória dos comandos recentemente utilizados).

2.2 Estrutura de Comandos

Alguns recursos da *C-SHELL* presentes nas subseções a seguir são dependentes do sistema operacional e refletem bem a versatilidade do sistema *UNIX*. Redirecionamento, *pipeline* de comandos, execução condicional e execução no *background* não são possíveis em muitos sistemas operacionais.

Os recursos que não são implicitamente dependentes do sistema, como variáveis, substituição de nome de arquivo, *alias*, *history*, estruturas de controle, etc poderiam ser implementados em qualquer sistema, melhorando em muito o poder de sua linguagem de interação com o usuário.

Na descrição da implementação (cap. 3) serão apresentados separadamente os recursos dependentes e independentes do sistema operacional.

2.2.1 Comandos Simples

Um comando simples é uma seqüência de palavras, sendo a primeira o nome do comando a ser executado e as restantes os seus argumentos. Após ler uma linha de comando e analisá-la a *shell* pesquisa o programa a ser executado na lista de nomes de arquivos contidos num dos diretórios especificados na variável do sistema *path*. Se encontrar o programa, cria um novo processo para sua execução para o qual passa argumentos especificados.

Uma lista de comandos é composta de um ou mais comandos simples separados pelo caracter “;” . Os comandos, neste caso, são executados seqüencialmente, não havendo desvios em caso de erro de algum comando.

Uma lista de comandos delimitada pelos símbolos “(” e “)” forma um grupo de comandos. Neste caso é criada uma nova instância da *shell* para a execução seqüencial destes comandos. Como os comandos são executados em uma outra instância da *shell*, as alterações provocadas no ambiente desta nova *shell* não irão influenciar a instância que originou a sua execução. Um grupo de comandos pode ser visto como um único comando, executado em uma outra *shell*.

2.2.2 Redirecionamentos

O meio mais comum de leitura é denominado dispositivo de leitura padrão e é geralmente direcionado para o terminal do usuário quando o interpretador inicia uma seção.

A função *getchar*¹, a cada vez que é chamada retorna o próximo caracter da entrada padrão. O terminal pode ser substituído por um arquivo usando o operador de redirecionamento “<”.

Ex. : Se *prog* utiliza a função *getchar*, ou alguma derivada desta, então a linha de comando :

```
% prog < nomearq
```

faz com que *prog* leia do arquivo *nomearq* em lugar do terminal. O programa *prog* não conhece as modificações ocorridas no ambiente de leitura, pois a C-SHELL providencia a troca nos descritores de arquivo.

¹todas as funções mencionadas nesta seção fazem parte da biblioteca padrão da linguagem “C”

Similarmente, *putchar(c)* coloca o caracter “c” no dispositivo de saída padrão, que é por *default* o terminal. A saída pode ser redirecionada para um arquivo usando o operador “>” .

Ex. : Se *prog* utiliza a função *putchar(c)*, ou alguma derivada desta, então a linha de comando:

```
% prog > nomearq
```

faz com que a saída de *prog* seja direcionada para o arquivo *nomearq* em lugar do terminal. Se o arquivo não existir ele é criado. Caso ele exista ele só será perdido (reescrito) se a variável do sistema *noclobber*² estiver desligada; caso contrário será emitida uma mensagem de erro e o programa não será executado. Mesmo estando ligada a variável *noclobber* é possível efetuar o redirecionamento: para isto é necessário utilizar ou o operador “>|”, que força o redirecionamento, ou o operador “>>”, que provoca a concatenação dos dados gerados pela execução do programa ao arquivo existente.

A saída padrão de erro também pode ser redirecionada, bastando para isto a concatenação do símbolo “&” após o símbolo “>” .

2.2.3 Pipeline de Comandos

Comandos simples separados pelo caracter “|” formam um *pipeline*. A saída padrão de um comando num *pipeline* é conectada à entrada padrão do próximo comando.

Um *pipe* é um canal para uso entre dois processos cooperantes: um processo escreve no *pipe* e o outro lê dele. O sistema controla o *buffer* de dados e sincroniza os dois processos.

2.2.4 Comandos no Background

Comandos simples seguidos pelo símbolo “&” são executados no *background*, i.e., eles são executados por processos concorrentes à *shell*. Quando um comando é colocado no *background* a *shell* não espera pelo seu término, mas sim executa outros comandos tão logo termine de criar o processo que executará o comando no *background*. Em alguns sistemas *UNIX* é possível retirar

²veja a seção 2.3.3, que trata do uso de variáveis na C-SHELL

um processo do *background* e executá-lo no *foreground*. Nestes sistemas a C-SHELL provê comandos para a execução deste recurso. Na implementação em discussão os comandos que retiram os processos do *background* não foram incluídos.

2.2.5 Execução Condicional

Comandos simples separados pelos símbolos “|” (OR) ou “&&” (AND) possuem a execução condicionada ao resultado da execução do comando anterior. O *status* de terminação dos comandos é conhecido e por meio dele pode-se determinar se a execução foi falha ou não.

- Significado semântico de *cmd1* | *cmd2* : o comando *cmd2* será executado se e somente se a execução do comando *cmd1* for falha.
- Significado semântico de *cmd1* && *cmd2* : o comando *cmd2* será executado se e somente se a execução do comando *cmd1* for normal.

2.3 Substituições na Linha de Comando

Assim que uma linha de comando é digitada ela é separada numa seqüência de palavras, a qual é colocada na lista *history* e em seguida analisada pela C-SHELL. Até este ponto nada difere a C-SHELL das demais linguagens de comando, pois várias delas possuem recursos para reedição de linha de comando.

Descreveremos a seguir várias transformações que a C-SHELL permite na linha de comando, na ordem em que elas são executadas. Algumas destas transformações são exclusivas das LCs do *UNIX*, como é o caso da substituição de comando, e outras exclusivas da C-SHELL, como é o caso da substituição *history* e *alias*.

2.3.1 History

A C-SHELL enumera todos os comandos executados, mantendo uma lista dos últimos comandos em memória ³.

³o número de comandos armazenados é especificado pelo valor da variável interna *history*.

A substituição *history* permite que palavras presentes nos comandos anteriores possam ser reaproveitadas como parte de novos comandos. Isto torna fácil a repetição de comandos, a repetição de argumentos de outros comandos no comando corrente, e até mesmo a correção de algum comando digitado incorretamente. Note que este mecanismo é muito diferente dos encontrados em outras LCs que possuem apenas recursos para repetição e edição de comandos anteriores.

Esta substituição é indicada pelo caracter “!” presente em qualquer lugar da linha de comando. Existe um grande número de operadores para execução da substituição *history* mas, por questões práticas, eles não serão aqui apresentados [SOB85]; Estes operadores depois de analisados são substituídos pelo resultado da operação. Antes de executar a linha de comando resultante da substituição a C-SHELL imprime a linha de comando na sua forma final na tela.

A decisão da não implementação deste tipo de substituição se deve ao fato do recurso *history* não ser comumente utilizado nos programas escritos em C-SHELL. Entretanto, nas versões interpretadas a sua implementação é imprescindível pois permite a reedição/repetição de comandos complexos de uma maneira simples e poderosa.

2.3.2 *Alias*

A C-SHELL permite a definição de uma lista de sinônimos para comandos. Este mecanismo pode ser usado para simplificar a digitação de comandos complexos e para o fornecimento de argumentos *default*. O uso de *alias* é semelhante ao uso de macros nas linguagens de programação tradicionais.

A lista de sinônimos é criada, listada e alterada pelos comandos *alias* e *unalias*

Exemplo : O comando `% alias ds ls -l`,

cria o comando *ds*, como um sinônimo para *ls -l*.

A substituição *alias* só é permitida na primeira palavra do comando: o sinônimos presentes no meio da linha de comando não serão substituídos.

2.3.3 Substituição de Variáveis

A C-SHELL mantém um conjunto de variáveis, cada uma das quais podendo possuir uma lista de zero ou mais palavras. Os valores destas variáveis podem ser vistos e modificados com o uso dos comandos *set* e *unset*, como será mostrado na seção que trata do uso de variáveis e expressões.

Após ter processado as substituições dos tipos anteriores (*history* e *alias*) a C-SHELL faz uma pesquisa na linha de comando pelo caracter "\$", que indica substituição de variáveis, ou seja, o nome da variável encontrado após este símbolo é substituído pelo seu valor.

Assim, o comando :

```
% echo $argv
```

irá imprimir a lista de argumentos passados na execução do arquivo de comandos (contida na variável padrão *argv*).

Como no mecanismo de *history*, existe um grande número de operadores que podem ser utilizados na substituição de variáveis. Alguns deles serão apresentados quando definirmos expressões com variáveis, os outros poderão ser encontrados em [SOB85] e em [JOY80].

2.3.4 Substituição de Comando

A substituição de comando é indicada pelo comando delimitado pelo caractere ` . O texto entre ` , que contém o comando e seus argumentos, será substituído pelo texto produzidos na saída padrão, após a execução do comando. Pode-se especificar com este recurso partes de um comando com a saída produzida pela execução de outros comandos.

Exemplo : O comando,

```
foreach i in ( `cat arquivo_de_dados | sort` )  
.....  
end
```

lê os dados do arquivo_de_dados de um modo ordenado, inserindo-os na lista de parâmetros do comando **foreach**.

2.3.5 Expansão de Nome de Arquivo

Este tipo de substituição provavelmente é o mais encontrado nas linguagens de comando. A C-SHELL fornece um conjunto de caracteres para serem utilizados na construção de expressões regulares para o casamento de padrões de nomes de arquivo.

Se um argumento da linha de comando contém qualquer um dos caracteres *,?,[,], ou começa com o caracter “~” ele sofre este tipo de expansão, a menos que este caracter esteja precedido pelo caracter de *escape* (“\”). Se um argumento contém algum destes caracteres e se o casamento deste padrão falhar, a C-SHELL assume que ocorreu um erro, imprime a mensagem *no match* e não executa o comando. Se a variável interna da C-SHELL, *nonomatch*, estiver ligada não será considerado erro a não existência de um arquivo cujo nome combine com o padrão especificado. Além disto a C-SHELL permite que a expansão de nome de arquivo seja inibida, caso a variável interna *noglob* tenha sido ligada.

Esta substituição é muito comum nas linguagens de comando. Existem algumas linguagens que apesar de aparentemente terem este recurso, a expansão é processada pelos programas aplicativos. A LC passa aos aplicativos o padrão encontrado na linha de comando. Isto provoca a replicação do mecanismo de substituição por todos os programas de aplicação que usam este recurso. Isso ocorre por exemplo com o COMMAND encontrado no PC-DOS e MS-DOS.

2.4 Expressões e Variáveis

Estruturas de controle e alguns tipos de substituições estão presentes em um grande número de linguagens de comando, o que não acontece com expressões e variáveis, presentes em muito poucas linguagens de comando. O uso de variáveis e expressões como nas linguagens de programação convencionais é que dão à C-SHELL flexibilidades não presentes na maioria das linguagens de comando.

A C-SHELL, como a BOURNE SHELL, possui somente variáveis do tipo cadeia de caracteres (*strings*), embora possa trabalhar com estas variáveis como se fossem números. Na BOURNE SHELL, para se avaliar uma expressão aritmética é necessário utilizar o programa *expr*; já a C-SHELL

possui “internamente” todas as funções presentes em *expr* e mais algumas outras. Estas funções para o cálculo de expressões lógicas e aritméticas são um subconjunto significativo das funções encontradas na linguagem “C”.

As expressões em C-SHELL podem aparecer nos comandos *set*, “@” (comando *set* para valores numéricos), *exit*, *if* e *while*. Os seguintes operadores estão disponíveis para o cálculo de expressões :

- ||, &&
- |, ^, &, !
- ==, !=, =~, !~, >=, <=, >, <, <<, >>
- ++, --
- +=, -=, *=, /=, %=, ^=, &=
- +, -, *, /, %
- ()

Os operadores ==, !=, =~, !~ tratam seus argumentos como *strings*; todos os outros operadores tratam seus argumentos como números. A semântica deles é a mesma encontrada na linguagem “C”, com exceção dos operadores =~ e !~, que tratam o segundo argumento da operação como um padrão, podendo conter qualquer um dos caracteres previstos para a formação de expressões regulares. Os operadores de atribuição (+=, -=, etc...) apesar de não existirem na C-SHELL original, foram incluídos nesta implementação.

São disponíveis ainda expressões da forma *-k nome_de_arquivo*, onde *k* é um das seguintes *flags*: *r*, *w*, *x*, *e*, *o*, *z*, *f*, *d* que indicam atributos ⁴ do arquivo dado como argumento.

O comando “@” deve ser utilizado quando uma expressão numérica for atribuída a uma variável. Deve-se recordar que em C-SHELL não existem variáveis numéricas explícitas, e internamente todas as variáveis são tratadas como *strings*.

Não existe, em C-SHELL, declaração de variáveis, sendo que a única limitação existente é de que variáveis indexáveis (vetores) devem ser iniciadas

⁴permissão para leitura, permissão para escrita, existência, etc..

antes de serem utilizadas. O exemplo a seguir ilustra o uso de variáveis do tipo vetor:

1. `% set source = (expr.c var.c path.c)`
2. `% echo $source`
`expr.c var.c path.c`
3. `% set source = ($source $source `date`)`
4. `% echo $source`
`expr.c var.c path.c expr.c var.c path.c oct 30,1988 10:30`

como pode ser visto no comando 3, uma vez definida uma variável como vetor, o número de elementos pode variar. Este comando também mostra como substituição de variável e de comando podem ser utilizadas em uma linha de comando.

Uma série de variáveis é utilizada internamente pela C-SHELL. Seus valores influenciam diretamente nas decisões do interpretador, como já foi visto no uso das variáveis *noclobber*, *nonomatch*, *noglob*, etc.

As principais variáveis internas da C-SHELL serão apresentadas a seguir:

- **argv** : Contém os argumentos passados na linha de comando.
- **\$n** : Parâmetro *n* da linha de comando.
- **history** : Controla o tamanho da lista dos últimos comandos digitados.
- **home** : Contém o *pathname* do diretório base ⁵.
- **path** : Lista de diretórios onde um arquivo executável é procurado.
- **status** : Contém o valor da condição de término de um programa.
- **echo** : Quando ligada, ⁶ faz com que a *shell* imprima a linha de comando antes de executar.

⁵*home directory*

⁶Uma variável é considerada ligada a partir do momento que foi referenciada pelo comando *set*. O comando *unset* desliga uma variável.

- **noclobber** : Quando ligada, impede a perda de arquivos em redirecionamentos.
- **noglob** : Quando ligada, impede a expansão de nome de arquivos.
- **nonomatch** : Quando ligada, impede a ocorrência do erro *no match* na expansão de nome de arquivo. A *shell* passa o padrão para o programa que está sendo chamado.

A lista completa destas variáveis e a função de cada uma delas pode ser encontrada em [JOY80].

2.5 Estruturas de Controle

As estruturas de controle da C-SHELL não diferem em muito das comumente encontradas nas linguagens de programação convencionais. Devido a isso, faremos somente um breve comentário sobre a sintaxe e a semântica dos comandos relacionados.

2.5.1 Controle do Fluxo

- **Comando IF** : A semântica deste comando é semelhante à encontrada na linguagem "C". Se *expr* for diferente de 0 o comando seguinte será executado, caso contrário será executado o comando da cláusula ELSE, se esta existir. A expressão *expr* pode ser qualquer uma das expressões aceitas pela C-SHELL.

Formas :

```
if ( expr ) comando
```

```
if ( expr ) then
    comandos
endif
```

```
if ( expr ) then
    comandos-1
else
```

```

        comandos-2
endif

if ( expr-1 ) then
        comandos-1
else if ( expr-2 ) then
        comandos-2
        ...
else
        comandos-n
endif

```

- Comando SWITCH : No comando *switch* o valor a ser comparado, *word*, deve ser uma *string* e os membros da cláusula *case* poderão ser padrões descritos por expressões regulares, compostas pelos símbolos discutidos na seção que trata de expansão de nome de arquivo. Tirando este detalhe, seu funcionamento é semelhante ao comando de mesmo nome da linguagem "C".

Forma :

```

switch ( word )
        case padrao-1 :
                comandos-1
        breaksw

        .....

        case padrao-n:
                comandos-n
        breaksw
        default :
                comandos-d
        breaksw
        endsw

```

- Comando GOTO : O rótulo *word* pode ser composto de caracteres de expansão de nome de arquivo ou uma substituição de comando. O

rótulo gerado deve ser uma *string* definida no programa do mesmo modo que são definidos os rótulos em "C".

Forma :

goto *word*

- Comandos **BREAK** e **CONTINUE** : A sintaxe e a semântica destes comandos é a mesma dos comandos de mesmo nome da linguagem "C".

Forma :

Comando Repetitivo
comandos
break ou **continue**
comandos
end do Comando Repetitivo

2.5.2 Comandos Repetitivos

- Comando **REPEAT**: Permite que uma linha de comando seja repetida um numero específico de vezes.

Forma :

repeat *expr_numerica* comando

- Comando **WHILE** : Possui semântica semelhante ao comando equivalente na linguagem "C".

Forma :

```
while ( expr )  
    comandos  
end
```

- comando **FOREACH** : Neste comando é realizada uma iteração para cada valor presente na lista de valores passada como argumento. A cada iteração *var*, assume o valor da lista passada como argumento.

Forma :

```
foreach var in ( lista_de_valores )  
    comandos  
end
```

2.6 Entrada e Saída de Dados

No C-SHELL original não existia um comando interno para processar entrada de dados, embora algumas implementações posteriores tenham implementado comandos para este fim; para saída é utilizado o comando *echo*. No presente compilador, foram implementados os comandos *read var* e *print expr* para tornar a comunicação com o usuário mais simples. Além disto, como o código é gerado em "C", o usuário pode incrementar a interface inserindo comandos no arquivo gerado.

Por razões práticas, uma série de comandos foram omitidos. Eles podem ser encontrados nas referências citadas anteriormente e no apêndice C, que contém o manual da primeira versão do compilador.

Capítulo 3

Implementação

Este capítulo é constituído de duas seções. Na primeira serão discutidos detalhes da implementação do compilador C-SHELL e na segunda serão apresentados detalhes da implementação da biblioteca.

3.1 Implementação do Compilador

3.1.1 Introdução

Por questões práticas se decidiu fazer a compilação em duas etapas. Na primeira etapa um programa escrito em C-SHELL é traduzido para a linguagem C e na segunda é compilado e ligado com a biblioteca do sistema e com a biblioteca do compilador. O esquema geral da implementação pode ser visto na figura 3.1.

A razão desta escolha se deve ao fato de que a linguagem C permite fácil acesso ao hardware e às primitivas do sistema *UNIX*. Deste modo, a atenção foi concentrada nos problemas de tradução de C-SHELL e não nos detalhes de geração de código em linguagem de máquina. Outro fator que motivou esta abordagem, foi o da portabilidade do código gerado. Com pequenas modificações na biblioteca que acompanha o compilador (“run time library”) pode-se utilizá-lo em várias versões do sistema *UNIX* ou similares. A versão atual do compilador é executável nos sistemas *UNIX* versão III e V.

Serão mostrados a seguir, alguns detalhes da implementação do com-

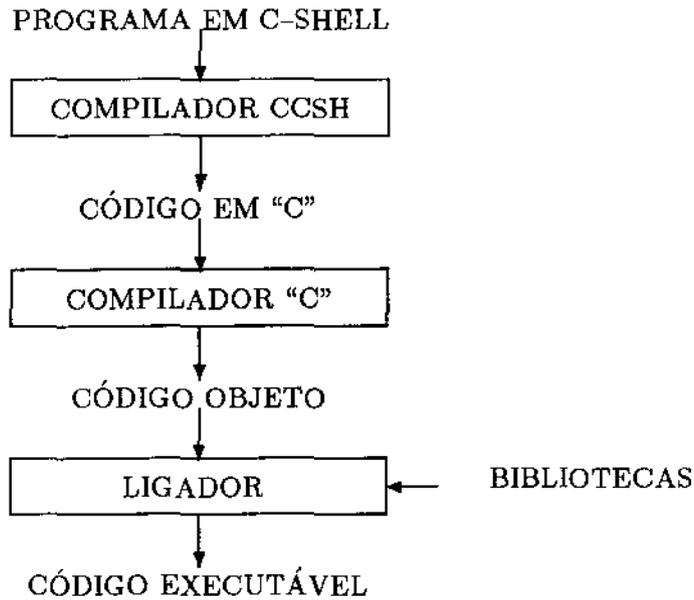


Figura 3.1: Esquema Geral do Processo de Compilação

compilador : análise léxica, análise sintática , recuperação de erros e geração de código. Na geração de código serão estudados detalhes das principais construções da C-SHELL e o seu código equivalente em linguagem C.

3.1.2 Análise Léxica

O analisador léxico corresponde ao nível mais baixo da estrutura do compilador. Ele é responsável pela conversão da seqüência de caracteres que compõem o programa fonte na seqüência de símbolos léxicos que irão alimentar o analisador sintático.

O analisador léxico do compilador C-SHELL foi construído usando o programa *Lex* [LES75] o qual é um gerador de analisadores léxicos.

O código fonte para o *Lex* é composto por uma tabela de expressões regulares e fragmentos de código em linguagem C, que correspondem às ações pertinentes. Para cada símbolo válido da linguagem, é especificada uma expressão regular para seu reconhecimento.

O programa gerado por *Lex* é um autômato de estados finitos determinista. Quando uma das expressões regulares que compõem o arquivo fonte do *Lex* é reconhecida, o fragmento de código associado é executado.

Estes fragmentos de código executam as tarefas necessárias para a análise léxica que vão além do simples reconhecimento de padrões (tratamento de identificadores, etc). Na maioria das vezes, estas ações correspondem ao retorno do valor que internamente representa o símbolo reconhecido.

A implementação de uma expressão regular para cada símbolo terminal da linguagem tem um grave problema: aumenta de modo considerável o número de estados do autômato. Um modo de contornar este problema é usar uma expressão regular que reconhece vários símbolos terminais de uma só vez, deixando para uma rotina auxiliar a determinação do código que deve ser retornado. Isto foi feito, com um ganho significativo na implementação do reconhecimento de palavras reservadas e identificadores. Este ganho foi representado pelo decréscimo de 50 por cento no tamanho do programa gerado. O *overhead* adicionado ao programa pela rotina de identificação foi compensado pela diminuição do número de estados intermediários do autômato.

O exemplo a seguir ilustra a utilização do gerador de analisadores léxicos. Este exemplo foi retirado do código fonte do compilador e ilustra a forma das expressões regulares e ações passadas para o *lex*.

EXPRESSÃO	AÇÃO
<code>\"([\n]* \\[\n])*\"</code>	<code>return (STR);</code>

Quando a expressão acima for reconhecida o comando *return(STR)* será executado.

3.1.3 Análise Sintática e Recuperação de Erro

O analisador sintático também foi gerado automaticamente a partir da ferramenta de software *Yacc*[JOH78].

A partir de uma linguagem de especificação que descreve a sintaxe desejada (gramática), o *Yacc* gera uma função capaz de reconhecer textos construídos segundo esta sintaxe. Esta função, chamada de *parser*, invoca uma rotina de entrada de nível mais baixo fornecida pelo analisador léxico para selecionar os itens básicos do texto (*tokens*) a ser analisado. A gramática utilizada como fonte para o *Yacc* deve estar no formato *LALR(1)*.

Como no *Lex*, a cada regra está associada um fragmento de código que é executado após o seu reconhecimento.

O *parser* gerado é um autômato a pilha constituído de: uma matriz de transição para derivar um novo estado a partir das combinações possíveis do estado corrente e do próximo símbolo da entrada; uma tabela de ações definidas pelo usuário; uma pilha; e uma rotina que utiliza estes dados para a análise.

A matriz de transição, os fragmentos de código que correspondem às ações e o interpretador são colecionados em uma função chamada *yyparse* que invoca repetidamente a função do analisador léxico (*yylex()*).

O estado corrente, no topo da pilha, e o próximo símbolo terminal produzido pela chamada da função *yylex()*, selecionam uma operação da matriz de transição. O arquivo *y.output* gerado pelo *yacc*, quando executado com a opção *-v*, mostra o conteúdo da matriz de transição para cada combinação de símbolo terminal e estado.

Uma situação de erro ocorre quando, para um determinado estado da matriz de transição, surge um símbolo inválido na entrada. Para este tipo de erro, o *yacc* gera um terminal *error* que pode ser utilizado na recuperação de erros como será mostrado a seguir.

Para ilustrar o tipo e a simplicidade do código passado para o *Yacc* usaremos um pedaço de uma definição de expressões:

Gramática :

```
expr : expr '+' expr
      | expr '-' expr
      | CONST
```

Código para o *Yacc*:

```
expr : expr '+' expr
      {
          soma();
      }
```

```

| expr '-' expr
{
    subtrai();
}
| CONST
{
    carreconst();
}

```

As ações *soma()*, *subtrai()* e *carreconst()* serão executadas após o reconhecimento da regra à qual ela está associada.

O esquema da integração do *Yacc* e do *Lex* pode ser visualizado na figura 3.2.

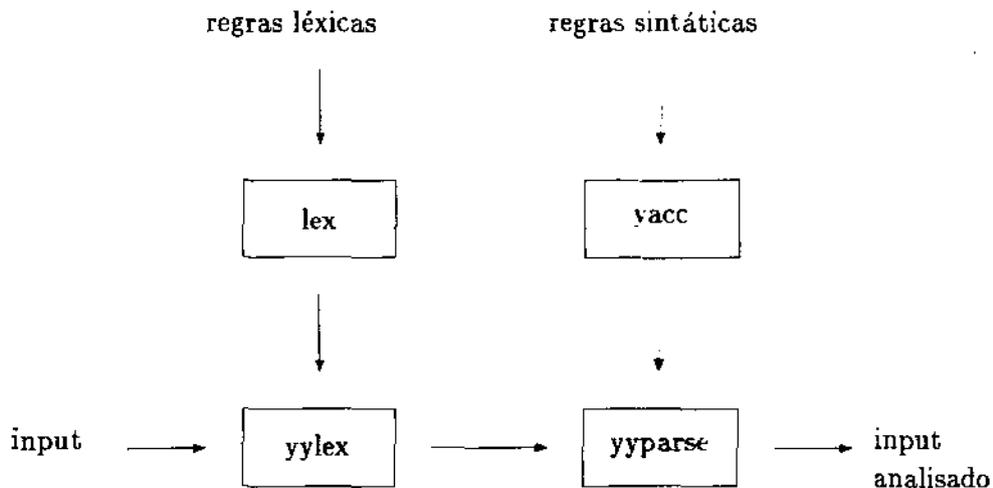


Figura 3.2: Esquema da Integração

A recuperação de erros de sintaxe (crítica e retoma à compilação) pode ser resolvida no *yacc* com relativa facilidade.

Basicamente existem dois modos de tratamento de erros:

- Tratamento a nível de símbolos, ou seja, adicionam-se regras “ilegais” na gramática para prever e “aceitar” os prováveis erros de sintaxe. Neste caso há sempre o perigo de se omitir algum erro.
- Tratamento do erro como um caso especial de um símbolo terminal. O *Yacc* fornece um símbolo terminal pré-definido (*error*) com esta finalidade, o qual pode ser usado como qualquer outro símbolo terminal definido pelo usuário. Uma vez ocorrida uma situação de erro, ele é gerado automaticamente.

Nos dois casos após o reconhecimento do erro, o usuário (programa aplicativo) deve tomar as devidas providências para sua recuperação.

Optou-se pela aplicação da segunda regra por ser mais segura e simples. O capítulo 4 de [SCH85] trata com detalhes este problema.

3.2 Implementação da Biblioteca do Compilador

3.2.1 Implementação da Parte Dependente do Sistema Operacional

Introdução

Para se prosseguir na apresentação do código gerado pelo compilador, é necessário uma pequena introdução a algumas primitivas do sistema operacional que controlam processos e o acesso a arquivos, através das quais foi implementado a maior parte dos comandos da C-SHELL dependentes do sistema.

Estas primitivas são :

- Controle de Processos :

fork() : Cria um novo processo. O processo criado (filho) é uma cópia física do processo que lhe deu origem (pai), diferindo apenas no valor retornado por esta primitiva. O processo pai recebe o número que identifica o filho (PID), e o filho recebe 0. O processo filho herda todos os arquivos abertos pelo pai.

exec(programa,lista-de-argumentos) : Substitui o código atual pelo código do programa passado como argumento e executa

o processo. Argumentos para o novo processo a ser executado devem ser passados como parâmetros na invocação.

exit (status) : Finaliza a tarefa com o *status* especificado.

wait (&status) : Esta primitiva faz um processo esperar pelo término de um processo filho. O *status* de terminação do processo filho é retornado no endereço passado como parâmetro.

kill(PID,sin) : Esta primitiva envia o sinal (mensagem pré-definida) *sin* para o processo cuja identificação é PID;

getpid() : Esta primitiva retorna o PID do processo que a chamou.

- Acesso a Arquivos :

open(nome,flag) : Esta primitiva abre (inicia para operação) o arquivo *nome* com tipo de acesso especificado por *flag* (escrita, leitura ou atualização). Retorna o descritor ¹ do arquivo aberto.

close (fd) : Fecha o arquivo cujo descritor é *fd*.

dup(descr) : Esta primitiva duplica o descritor especificado, retornando o primeiro descritor de arquivo disponível. Os descritores, novo e velho, compartilham o mesmo ponteiro no arquivo (*file pointer*).

pipe(descrs) : Esta primitiva retorna dois descritores de arquivo, um para escrita (*descrs[1]*) e outro para leitura (*descrs[0]*). Os dados são transmitidos através do *pipe* segundo uma disciplina de fila circular: nenhum dado pode ser lido duas vezes e nenhum dado é escrito sobre outro existente.

chmod(arq,perm) : Muda os modos de acesso possíveis do arquivo.

chdir(dir) : Muda o diretório corrente de uma tarefa.

- Rotinas Auxiliares :

dofmatch(dir,pat,vet) : Esta rotina retorna no vetor *vet* todos os nomes de arquivo do diretório *dir* que “casam” com o padrão *pat*.

match(string,padrão) : Pesquisa um padrão numa cadeia.

times(buffer) : Tempo de execução da tarefa.

¹estrutura de dados que identifica um arquivo

`getenv(nome)` : Obtém o valor de uma variável do sistema.

Além destas, foram utilizadas uma série de rotinas que fazem parte da biblioteca do sistema. Em [DIG87] existe uma lista detalhada destas funções.

De posse destas primitivas, foi desenvolvido um conjunto de rotinas que compõem atualmente a biblioteca do compilador, sendo as principais listadas no apêndice B.

No sistema Unix, a entrada padrão (*stdin*) está sempre associada ao descritor 0, a saída padrão (*stdout*) ao descritor 1 e a saída de erro padrão (*stderr*) ao descritor 2. Estes valores serão utilizados nas subseções a seguir na demonstração de como foram implementados os recursos de redirecionamento, *pipeline* e substituição de comandos.

Implementação de Redirecionamento de E/S

O redirecionamento de E/S foi implementado através do uso das primitivas *open*, *close* e *dup* do sistema, combinadas de modo conveniente.

- Redirecionamento da Entrada Padrão : Para se fazer o redirecionamento de entrada basta abrir o arquivo de entrada para leitura, fechar a entrada padrão do sistema (*stdin*) e duplicar o descritor do arquivo aberto.

A figura 3.3 mostra o algoritmo utilizado no redirecionamento de entrada padrão.

Ao fechar o arquivo de entrada padrão, o descritor 0 é liberado; ao se duplicar o descritor do arquivo a ser utilizado como entrada, o descritor 0 passou a compartilhar o mesmo *file pointer*. Com isto, todas as leituras que eram realizadas utilizando o descritor 0 passaram a ser realizadas no arquivo. Isto é sempre válido pois a primitiva *dup* retorna o menor descritor disponível, no caso, o descritor 0. Este comentário também é válido para o redirecionamento da saída padrão.

- Redirecionamento da Saída padrão : Segue os mesmos procedimentos do redirecionamento da entrada, só que se fecha a saída padrão e o arquivo é aberto com a opção de escrita.

A figura 3.4 mostra o algoritmo utilizado para implementar redirecionamento de saída.

- Redirecionamento do Erro Padrão : tem uma implementação idêntica ao redirecionamento da saída padrão, bastando trocar a constante OUT pela constante ERR (erro padrão), como parâmetro da primitiva *close*.

O redirecionamento através do operador “>>” (*append*) é implementado de maneira análoga ao redirecionamento da saída padrão. A única diferença é que o arquivo é aberto com a opção O_APPEND, constante que define abertura de arquivo para concatenação.

Implementação de *Pipeline* de Comandos

Esta implementação foi feita por intermédio da primitiva *pipe* do sistema. Como foi mostrado, esta primitiva retorna dois descritores de arquivo, um para leitura e outro para escrita.

O algoritmo padrão para implementação de *pipes* de comandos é mostrado na figura 3.5.

Para implementação do pipeline de comandos, o processo pai executa a primitiva *pipe*, criando um canal de comunicação. O descritor 1 é usado para escrita e o descritor 0 é usado para leitura no canal. O processo pai cria então os dois processos filhos, que herdam os descritores de arquivo abertos pelo pai, inclusive os retornados pela primitiva *pipe*. Por uma convenção pré-estabelecida, um filho irá produzir dados e escrevê-los no canal de comunicação, enquanto que o outro, irá consumir dados provenientes do canal.

Se um processo tenta ler de um *pipe* vazio será automaticamente colocado no estado *sleeping*, permanecendo inativo até que um dado chegue. Se o outro processo tentar escrever no *pipe* e ele estiver cheio, será colocado no estado *sleeping* até que se tenha área disponível para escrita no *buffer*. Quando o lado da escrita do *pipe* é fechado, a leitura subsequente retornará o valor EOF (caracter de fim de arquivo).

```

fd = open (nome_arq_entrada, O_RDONLY);
if ( fork() == 0) {
    /* código do filho */
    close(IN); /* fecha stdin */
    dup(fd); /* duplica o descritor fd. Como
dup retorna o menor descritor: stdin == fd */
    close(fd) ; /* fecha fd que não é mais necessário */
    /* programa do filho */
}
wait(& status);

```

Figura 3.3: Esquema de Implementação de Redirecionamento de Entrada Padrão

```

fd = open(nome_arq_saida, O_WRONLY | O_CREAT);
if( fork() == 0) {
    /* código do filho */
    close (OUT); /* fecha a saída padrão */
    dup(fd); /* stdout == fd */
    close(fd); /* fecha fd que não é mais necessário */
    /* programa do filho */
}
wait(&status);

```

Figura 3.4: Esquema de Implementação de Redirecionamento de Saída Padrão

```

pipe(&fd[0]); /* cria um pipe */
filhoA = fork(); /* cria o processo A */
if( filhoA == 0 ){
    /* filhoA continua aqui */
    close(OUT); /* fecha o descritor para a saída padrão */
    dup(fd[1]); /* duplica o descritor fd[1] */
    /* processo A escreve no PIPE */
    exec(progA,Lista_de_ParametrosA);
}
else{
    /* o processo pai continua aqui */
    filhoB = fork();
    if( filhoB == 0 ){
        /* filhoB continua aqui */
        close(IN); /* fecha o descritor para
a entrada padrão */
        dup(fd[0]); /* duplica o descritor fd[0] */
        /* o processo B lê do PIPE */
        exec(progB,Lista_de_ParametrosB);
    }
    else{
        /* o processo pai continua aqui */
        /* fecha os descritores do PIPE
e espera pelos filhos */
        close(fd[0]);
        close(fd[1]);
        idfilho1 = wait(&status); /* espera um filho */
        idfilho2 = wait(&status); /* espera o outro filho */
    }
}
/* o processo pai continua aqui após a morte dos filhos */

```

Figura 3.5: Esquema da Implementação de *Pipeline* de Comandos

Comandos no *Background*

O sistema UNIX permite que um usuário dispare vários processos assíncronos. A linguagem de comando que promove a interface do usuário com o sistema, por conseguinte, tem que explorar esta característica do sistema, fornecendo aos usuários um modo de usar este recurso.

Em C-SHELL, isto é feito através do separador de comandos "&". A execução de comandos no *background* é implementada como mostra a figura 3.6.

```
if ( fork() == 0 ){
    /* código para o filho */
    execv(programa,Lista.de.Parametros);
}
/* código para o pai, o processo pai pode continuar
sua execução, inclusive disparando a execução de outros
processos filhos */
```

Figura 3.6: Esquema da Implementação de Comandos no *Background*

Note que, neste caso, o processo pai não executa a primitiva *wait*, responsável pela sincronização do pai com o término da execução do filho. Isto também ocorre em comandos que fazem parte de um *pipeline*.

Substituição de Comandos

Este tipo de substituição em linha de comando é, sem dúvidas, o mais complexo que existe em C-SHELL e o que exigiu maior cuidado na implementação.

O esquema de execução de substituição de comando pode ser visto na figura 3.7.

A substituição de comando foi também implementada a partir da primitiva *pipe* do sistema. O esquema da implementação é o seguinte: o processo pai cria um processo filho (comando a ser substituído) que irá produzir dados no canal de comunicação criado pela primitiva *pipe*. Em seguida, o processo pai fecha o seu arquivo de entrada padrão e começa a ler dados do

canal de comunicação, montando um vetor de *strings*, que será o resultado da substituição do comando. Para terminar a operação, ele restaura a sua entrada padrão e retorna ao processo que estava executando.

```
pipe(&fd[0]);
fdo=fp{OUT};
if( fork() == 0){
    /* código para o filho - processo produtor */
    close(OUT);
    dup(fdo);
    close(fdo);
    close(fp{IN});
    exec(comando,Lista.de.Parametros);
}
else{
    /* código para o pai - processo consumidor */
    close(fdo);
    fdi = fp{in};
    close(IN);
    dup(fdi);
    k = 0;
    while( (i=getchar()) != EOF && i != NULL)
        if( i != ' ' && NO_TAB(i))
            s[k++] = i;
        else{
            s[k] = NULL;
            strcpy(arg{n.arg++},s);
            k = 0;
        }
}
```

Figura 3.7: Esquema da Implementação de Substituição de Comandos

O vetor de argumentos é montado com valores lidos do processo filho (comando que está sendo substituído). Este esquema de execução é *seguro*, ou seja, nenhum dado produzido pelo filho será perdido pelo processo pai, pois como foi visto, a primitiva *pipe* cria um canal *seguro* de comunicação entre processos. Quando um processo consumidor tenta ler um dado ainda não disponível, é colocado em estado suspenso e só é retirado quando houver dado para ser lido. Do mesmo modo, se o processo produtor tenta produzir um dado e não existir área para escrita, ele é suspenso e fica aguardando a

liberação de área pelo processo leitor.

- Programa em C-SHELL para exemplificar a implementação de substituição de comando.

```
#
# demonstracao de substituicao de comando
#

# teste de substituicao de comando em expressoes
set a = ( 'cat word' 'ls *.c' )

# teste de substituicao de comando em linha de comando
cc -c 'ls *.c'
```

Código gerado para substituição de comando. Por questões práticas foram eliminadas as linhas de iniciação e declaração do código gerado.

```
1-  push_id("a",INSERT);
2-  inicio_dcl_vetor(); /** marca que e inicio de vetor **/
    iarg = 0;
    strcpy(arg[iarg++],"cat");
    strcpy(arg[iarg++],"word");
3-  subs_comando(0,&iarg,arg); /** executa a substituicao do **/
4-  ins_vet(arg,iarg,STRING); /** comando 1 **/
    iarg = 0;
    strcpy(arg[iarg++],"ls");
5-  expande_n_arquivo(".", "*.c",arg,&iarg);
6-  subs_comando(0,&iarg,arg); /** executa a substituicao do **/
7-  ins_vet(arg,iarg,STRING); /** comando 2 **/
8-  dcl_vetor(STRING); /** termina a declaracao de vetor **/

    iarg=0;
    strcpy(arg[iarg++],"cc");
    strcpy(arg[iarg++],"-c");
    n_arg = iarg;
    strcpy(arg[iarg++],"ls");
9-  expande_n_arquivo(".", "*.c",arg,&iarg); /** expande nome de arq. **/
10- subs_comando(n_arg,&iarg,arg); /** executa a substituicao **/
    executa(NREDIN,NREDOUT,NPIPE_A,NPIPE_D,NBACK,"","");
```

Análise do código gerado :

1. Pesquisa o descritor da variável "a".
2. Marca início de declaração de vetor.
3. Executa a substituição de comando. O corpo desta rotina pode ser visto na figura 3.7.
4. Copia as *strings* geradas pela substituição de comando no vetor "a".
5. Expande o padrão "*.c" no diretório corrente (".").
6. Idêntico a 3.
7. Idêntico a 4.
8. Fecha a declaração de vetor.
9. Idêntico a 5.
10. Idêntico a 6.

Substituição de Nome de Arquivo

Quando o compilador detecta um caracter de expansão de nome de arquivo (*, ?, []), ele invoca a rotina do sistema *dofmatch* que pesquisa a existência de arquivos cujo nome combina com o padrão especificado, devolvendo uma lista destes. Se a variável da C-SHELL *noclobber* estiver ativa (ON), nenhuma expansão é realizada.

Apesar de ter sido implementada utilizando uma rotina do sistema, a substituição de nomes de arquivos poderia ter sido implementada de modo independente do sistema. Para isto seria necessário somente a implementação de um algoritmo de casamento de padrões. Alguns destes algoritmos podem ser encontrados em [SED84].

Código gerado para expansão de nome de arquivos:

A linha 5 do exemplo anterior mostra a chamada à rotina da biblioteca que expande nome de arquivo. Os argumentos passados são : o diretório onde será executada a expansão, o padrão a ser utilizado no casamento, um ponteiro para o buffer de armazenamento dos arquivos expandidos, e o índice a partir do qual serão realizadas as inserções no vetor. Esta rotina utiliza a função de biblioteca *dofmatch*, apresentada no início deste capítulo.

3.2.2 Implementação da Parte Independente do Sistema Operacional

Variáveis

O uso de variáveis como em uma linguagem de programação, é sem dúvida, uma das grandes vantagens da C-SHELL que poderia ser incorporada na maioria das linguagens de comando.

Todo o gerenciamento de memória para variáveis é dinâmico, pois como já foi mencionado, as variáveis podem trocar de tipo e dimensão durante a execução do programa compilado. Nos compiladores em que existe declaração de variáveis e tipos “fortes” (*strongly typed*), o espaço de cada variável é alocado durante a compilação, ou seja, são alocadas posições fixas na pilha de execução ou no segmento de dados para cada variável. Na implementação em estudo, a tabela de símbolos permanece durante a execução do código gerado, cada variável ocupando uma entrada na tabela.

A tabela de símbolos é representada pelas estruturas de dados mostradas na figura 3.8.

```
struct SIMBOLO {
    unsigned char nshift; /* num. de shifts em vetores*/
    int nelelem; /* num. de elementos em vetores*/
    unsigned char onoff:1, /* variável está on ou off */
    tipo:2, /* tipo da variável : int. ou str. */
    categ:2, /* simples ou vetor */
    union{
        int val ; /* valor de variáveis inteiras */
        char *str; /* ponteiro para strings */
        struct {
            struct CHUNK *vet; /* ponteiro para a
                estrutura que contém os elementos do vetor */
            struct CHUNK *prim; /* ponteiro para o prim. elemento */
        }vetor;
    }valor;
}tab_simbolo[MAXVAR];
```

Figura 3.8: Estrutura de Dados para a Tabela de Símbolos

O campo *nshift* é utilizado para a implementação do comando *shift*, que provoca o deslocamento dos elementos dentro do vetor. O conteúdo do campo corresponde ao número de elementos saltados a partir do início do vetor. Os valores deslocados não são perdidos, eles podem ser recuperados pelo comando *unshift*².

O campo *onoff* indica se uma variável está ou não ativa em um determinado instante. O comando *unset VAR* desliga este campo, sendo o conteúdo da variável preservado. Os campos *tipo* e *categ* determinam o modo da representação interna da variável.

O valor da variável, dependendo do seu tipo interno e da sua categoria, irá ocupar um dos campos da *union valor*. O campo *val* é utilizado para armazenar valores das variáveis representadas internamente como inteiros; o campo *str* é utilizado para armazenar ponteiros para *strings* e o campo *vet* para armazenar ponteiros para a estrutura de vetores.

A estrutura que implementa vetores é mostrada na figura 3.9.

```
struct CHUNK{
    unsigned nelem; /* número de elementos */
    struct CHUNK *next; /* ptr. para o próximo chunk */
    struct CHUNK *prev; /* ptr. para o chunk anterior */
    struct CELL elem[TAMCHUNK]; /* espaço para armazenar os elementos do vetor */
};
```

Figura 3.9: Estrutura de Dados para a Implementação de Vetores

Como a dimensão do vetor não é conhecida em tempo de compilação, a alocação dos *chunks* (blocos) de memória é processada por demanda. Quando se esgota o espaço de armazenamento disponível em um *chunk* é solicitado um novo à rotina de alocação. Esta rotina verifica se existe algum na lista de disponíveis, e retorna o próximo *chunk* disponível. Caso a lista esteja vazia, um novo *chunk* é solicitado ao sistema operacional. O *chunk* retornado é então acoplado ao anterior pelo campo *next* e *prev*, formando uma lista duplamente ligada com os elementos do vetor.

²este comando não existe na C-SHELL original

O campo *vet* da estrutura SIMBOLO aponta para o primeiro *chunk* alocado para o vetor; o campo *prim* aponta para o *chunk* que contém o primeiro elemento do vetor ³.

O elemento *i* do vetor ocupa a posição $(nshift+i)\%TAMCHUNK$ no *chunk* de número $(nshift+i)/TAMCHUNK$.

Para a implementação dos comandos *shift* e *unshift* é necessário ajustar o ponteiro *prim*, de modo que ele fique sempre apontando para a primeira posição do *chunk* que contém o primeiro elemento do vetor. É também verificado se o número de *shifts* é maior que o número de elementos. Caso isto ocorra, será apresentada uma mensagem de erro, e o programa será abortado.

Cada uma das células que compõem um elemento do vetor pode conter um inteiro ou um ponteiro para um vetor de caracteres como mostra a figura 3.10.

```
struct CELL {
    int valor;
    char *str;
};
```

Figura 3.10: Estrutura de Dados das Células

É ainda necessário mencionar que o tipo e a classe internos de uma variável são ditados pelo comando usado na sua definição. O comando *set* define variáveis do tipo *string* e o comando "@" variáveis inteiras.

Expressões

O uso de expressões em C-SHELL segue os mesmos fundamentos das linguagens de programação convencionais, mas como esta linguagem manipula objetos abstratos, ela possui operadores especiais que não estão normalmente presentes em linguagens de programação.

³note que o primeiro elemento do vetor pode não estar no primeiro *chunk*, isto vai depender do número de comandos *shift* já efetuados.

Para a avaliação de expressões foi utilizada uma pilha da forma da figura 3.11.

```
struct STACK_ENTRY {
    char tipo; /* tipo do elemento da pilha */
    union{
        int valor;
        SIMBOLO *descr;
        char *str;
    }dado;
} pilha[TAMPILHA];
```

Figura 3.11: Estrutura de Dados para a Implementação da Pilha de Execução

O campo *tipo*, indica se o elemento é uma *string*, um valor inteiro ou um ponteiro para a tabela de símbolos onde está armazenado uma variável.

O código gerado pelo compilador para o cálculo dos operadores é constituído de chamadas às funções do módulo de biblioteca *expr.o*.

No cálculo de expressões aritméticas em que um dos operadores é uma *string*, se a conversão para inteiro for possível, será executada, senão a expressão será calculada como se o valor deste operando fosse 0. Para evitar que estas conversões fossem executadas com grande frequência se optou pela diferenciação interna do tipo das variáveis. Uma variável definida pelo comando "@" é marcada internamente como inteira, embora para o usuário não seja imposta nenhuma restrição, ou seja, a nível de programação isto é transparente: não existem diferenças entre a utilização das variáveis na C-SHELL interpretada e compilada.

Os operadores relacionais especiais == e != atuam sobre *strings* e são implementados através da rotina *strcmp* da biblioteca padrão da linguagem C. Nos operadores =~ e !~ o segundo operando pode ser um padrão e portanto exige a utilização de funções mais complexas. No sistema *UNIX*, estas funções fazem parte da biblioteca do sistema, facilitando assim a implementação deste tipo de expressão.

Para implementação das expressões com arquivos foi utilizada a rotina

stat da biblioteca do sistema. Neste caso, como as informações armazenadas sobre os arquivos dependem da implementação do sistema de arquivos, é possível que algumas das *flags* implementadas para o sistema *UNIX* não sejam possíveis de implementar em outros sistemas. Por outro lado, o sistema pode armazenar outras informações, que tornariam viáveis outras expressões sobre os atributos do arquivo.

Controle do Fluxo

Com o objetivo de tornar a compreensão mais simples e objetiva, optamos nesta seção, pela apresentação do código gerado pelo compilador para cada umas das construções de controle de fluxo vistas no capítulo anterior.

Programa exemplo escrito em C-SHELL

```
#
# demonstracao do codigo gerado para os comandos
# while, if, switch, repeat e foreach
#

# comando while
@ a = 10
while ( $a > 0)
    echo "conteudo de a : " $a
end

# comando if
if( $a > $c ) then
    echo "o valor de a e maior que o de c"
else
    echo "o valor de c e maior ou igual ao de a"
endif

# comando switch
switch ( $a)
    case ab*cd :
        echo "casou padrao 1"
        breaksw
    case efg.* :
        echo "casou padrao 2"
        breaksw
endsw

# comando repeat
```

```
repeat $a echo "aguarde mais um pouco"

# comando foreach
foreach a ( teste1 teste2 teste3 teste4)
    echo $a
end
```

Código Gerado Para o Programa Exemplo

Obs: Os comentários não são gerados pelo compilador, mas foram inseridos para facilitar o entendimento do código.

```
/******definições de constantes,tipos e variáveis globais*****/

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include "/usr/include/ccsh/var.h"
#include "/usr/include/ccsh/ccsh.h"
#include "/usr/include/ccsh/var.e"
#include "/usr/include/ccsh/expr.e"
#include "/usr/include/ccsh/path.e"

main(argc,argv)
int argc;
char *argv[];
{
    /****** variáveis locais *****/

    int i,j,k,fdi,fdo,fp[2],cnt,n_arg,stat=0;
    char buffer[MAXARGS][MAXTAMARG];
    char path[64].*s,*ccstr;

    /***** inicia variáveis internas da C-SHELL *****/
    inicvar(argc,argv);

    /***** cria tabela de hash para nome de arquivos *****/
    create_tab();

    /***** inicia tabela de alias ****/
    cria_tabela_alias();

    for(i=0;i<MAXARGS;i++) arg[i]=buffer[i];

    /** comando WHILE **/
```

```

push_id("a",INSERT);    /** carrega endereco de "a" **/
car_int(10);            /** carrega valor de 10 **/
atribuicao(INTEIRO);    /** atribui a = 10 **/
while (1) {
    push_id("a",INSERT);
    push_valor();      /** pega valor de a **/
    car_int(0);        /** carrega valor 0 **/
    maior();           /** compara se maior **/
    if (!retira_topo()) break ; /** termina laço **/
    iarg=0;            /** se falso **/
    strcpy(arg[iarg++],"echo"); /** carrega programa**/
    strcpy(arg[iarg++],"conteudo de a :");/** arg1 **/
    push_id("a",INSERT);
    push_valor();
    subs_var(arg,&iarg); /** arg2 = $a **/
    /* executa programa **/
    executa(NREDIN,NREDOUT,NPIPE_A,NPIPE_D,NBACK,"","");
}

    /** comando IF **/

push_id("a",INSERT);
push_valor();
push_id("c",INSERT);
push_valor();
maior();                /** executa $a > $c **/
if (retira_topo()) {   /** o resultado da comp. e' **/
    iarg=0;            /** deixado no topo da pilha **/
    strcpy(arg[iarg++],"echo");
    strcpy(arg[iarg++],"o valor de a e maior que o de c");
    executa(NREDIN,NREDOUT,NPIPE_A,NPIPE_D,NBACK,"","");
}
else {
    iarg=0;
    strcpy(arg[iarg++],"echo");
    strcpy(arg[iarg++],"o valor de c e maior ou igual ao de a");
    executa(NREDIN,NREDOUT,NPIPE_A,NPIPE_D,NBACK,"","");
}

    /** comando SWITCH **/

push_id("a",INSERT);
ccstr = push_valor(); /** calculo da expr. $a **/
if(exp_reg("ab*cd",ccstr)){ /** testa padrao 1 **/
    iarg=0;
    strcpy(arg[iarg++],"echo");
}

```

```

    strcpy(arg[iarg++], "casou padrao 1");
    executa(NREDIN, NREDOUT, NPIPE_A, NPIPE_D, NBACK, "", "");
}
else
    if(exp_reg("efg.*", ccstr)){ /** testa padrao 2 **/
        iarg=0;
        strcpy(arg[iarg++], "echo");
        strcpy(arg[iarg++], "casou padrao 2");
        executa(NREDIN, NREDOUT, NPIPE_A, NPIPE_D, NBACK, "", "");
    }

        /** comando REPEAT **/

push_id("a", INSERT);
push_valor();
subs_var(arg, &iarg);
cnt = retira_topo(); /** num. de vezes da repeticao **/
while ( cnt-- > 0){ /** executa o loop **/
    iarg=0;
    strcpy(arg[iarg++], "echo");
    strcpy(arg[iarg++], "aguarde mais um pouco");
    executa(NREDIN, NREDOUT, NPIPE_A, NPIPE_D, NBACK, "", "");
}

        /** comando FOREACH **/

ccnel = 0;
car_str("teste1");
car_str("teste2");
car_str("teste3");
car_str("teste4");
car_arg(&ccnel); /** carrega lista de argumentos **/
for(ccn=0; ccn<ccnel; ccn++){ /** repete a iteracao **/
    push_id("a", INSERT); /** para cada um dos **/
    car_str(ccaux[ccn]); /** argumentos **/
    atribuicao(String);
    iarg=0;
    strcpy(arg[iarg++], "echo");
    push_id("a", INSERT);
    push_valor();
    subs_var(arg, &iarg);
    executa(NREDIN, NREDOUT, NPIPE_A, NPIPE_D, NBACK, "", "");
}
}

```

Substituições : Alias e de Variáveis

- Código gerado para *Alias* : A verificação de *Alias* é realizada pela rotina `ver_alias` presente no módulo `parserlc`. Esta rotina faz uma pesquisa na tabela de *hash* pela ocorrência da *string* passada como argumento na chamada da função. Se encontra a *string*, a função retorna o sinônimo correspondente, caso contrário, retorna *NULL*. As inserções na tabela de *Alias* (causadas pelo comando *alias*) são realizadas pela rotina `define_alias` e as eliminações (comando *unalias*) são realizadas por `unalias`.
- Código gerado para substituição de variáveis: Este tipo de substituição foi ilustrado em alguns dos exemplos anteriores. Faremos agora uma breve apresentação das rotinas de manipulação de variáveis.

O código gerado para `$echo $a` é:

```
iarg=0;
strcpy(arg[iarg++], "echo");
push_id("a");
push_valor();
subs_var(arg, &iarg);
executa(NREDIN, NREDOUT, NPIPE_A, NPIPE_D, NBACK, "", "");
```

`push_id` : Esta rotina carrega o descritor da variável "a" no topo da pilha.

`push_valor` : Esta rotina carrega o valor da variável cujo descritor está em `pilha[topo]`, no topo da pilha.

`subs_var` : Esta rotina carrega o vetor `arg` com o valor que está no topo da pilha.

Capítulo 4

Conclusão

4.1 Introdução

Um anúncio de um compilador para a BOURNE SHELL que tem aparecido com frequência nos últimos meses em revistas especializadas em UNIX (e.g. *Unix World*), resume em poucas palavras as vantagens da compilação dos programas escritos em linguagem de comando (*shell scripts*). Este anúncio diz o seguinte : “Torne seus programas escritos em linguagem de comando mais rápidos”.

O compilador anunciado possui as mesmas características do apresentado nesta dissertação. Ele gera código em linguagem “C” para posterior compilação com compiladores existentes. A semelhança na abordagem da implementação e o fato do compilador ser para a BOURNE SHELL, uma linguagem com menos recursos que a C-SHELL, nos motivaram a aprofundar ainda mais o nosso estudo e tentar conseguir resultados melhores que os obtidos nesta implementação e que serão apresentados na próxima seção.

Dentre as vantagens encontradas da compilação de linguagens de comando, destacam-se as seguintes:

- Os programas escritos em CCSH (C-SHELL compilada) são mais rápidos, uma vez que estes são compilados e os programas escritos nas outras linguagens são interpretados.
- Uma vez terminada a implementação do compilador em outros sistemas, os programas em CCSH serão totalmente portáveis. Hoje, o

sistema gera código para UNIX versão III e está sendo implementada a versão para UNIX versão V. A implementação do compilador em outros sistemas *unix-like* se resume a alterações nas funções da biblioteca. Na maior parte dos casos o compilador não irá sofrer modificações pois foi implementado utilizando apenas funções da biblioteca padrão da linguagem "C".

- O código gerado é de fácil entendimento, o que permite otimizações manuais. Estas otimizações poderão ser inseridas diretamente no código gerado. A seguir será mostrado um exemplo bem simples de como isto pode ser feito :

Suponha que o usuário necessite executar o comando :

```
$ translit A-Z a-z < file
```

Para a execução deste comando a *shell* terá que criar um novo processo e em seguida pesquisar pelo *path* do comando na tabela de *hashing*. Como este *path* é previamente conhecido, o trabalho de busca pode ser evitado. Para isso, é necessário inserir o valor conhecido no código gerado e retirar a chamada à rotina de pesquisa pelo *path*, que está embutida na função *executa()*. Esta é uma otimização bem simples, mas que pode ser de grande valia, pois na maior parte das vezes os aplicativos estão em uma área fixa do sistema de arquivos.

Outra vantagem de se ter código em "C" foi vista na última seção do capítulo 2, onde foi discutida a necessidade de uma entrada e saída mais elaborada.

- Há um ganho em segurança, pois o código final é um arquivo binário executável. Num arquivo executável é possível utilizar o mecanismo de *set user id* que permite um controle mais eficaz e inviolável dos acessos e direitos do programa. As versões interpretadas utilizam arquivos *ASCII* que são facilmente violáveis. Isto é particularmente interessante para programas administrativos.
- Dado ao alto nível da C-SHELL é possível utilizá-la para prototipagem de programas. Programas complexos podem ser desenvolvidos a partir da reutilização de programas existentes, ou a partir da combinação destes com outros desenvolvidos pelo usuário. O exemplo mostrado a seguir foi retirado da revista *CACM* de maio de 1985 página 456, e ilustra bem o reaproveitamento de programas para desenvolvimento de protótipos. Trata-se de um programa para correção ortográfica e

segundo o autor levou uma tarde para ser feito. Um programa para correção ortográfica pode ser visto do seguinte modo:

prog1 lê o arquivo texto fornecido e o divide em palavras.
prog2 ordena as palavras, eliminando as palavras duplicadas.
prog3 compara as palavras com um dicionário e lista as diferenças encontradas.

O código deste programa em C-SHELL, utilizando algumas das ferramentas de *software* existentes no *UNIX* é o seguinte :

```
# remove caracteres de formatação
prepare filename          |
#transforma maiúsculas em minúsculas
translit A-Z a-z         |
#remove pontuações
translit ^ a-z @n        |
#ordena as palavras
sort                      |
#remove as palavras duplicadas
unique                    |
#compara com o dicionário
common -2 dicionario
```

4.2 *Benchmarks* - Análise de resultados obtidos

Devido à falta de um interpretador para a linguagem C-SHELL em nossa instalação a comparação foi processada utilizando um interpretador para a BOURNE SHELL. Procurou-se utilizar recursos equivalentes nas duas linguagens de modo a minimizar distorções nos resultados.

O tempo de execução dos programas foram medidos pelo utilitário *time*. Este utilitário executa o comando passado como argumento e exhibe o tempo real, o tempo de cpu e o tempo de sistema. O tempo real é o tempo total decorrido entre o início da execução do comando até o seu término. O tempo de cpu é o tempo efetivamente gasto na execução do comando e o tempo de sistema é o tempo de cpu que o sistema usou para processar o comando.

A precisão dos tempos reportados é a do relógio de *hardware*, ou seja, décimos de segundo.

Os dados apresentados a seguir foram medidos em um D8000 da Digirede.

TESTE	CCSH			ISH		
	total	cpu	sis	total	cpu	sis
1	4.0	0.1	0.2	5.0	0.5	3.3
2	2:31.0	4.9	18.3	2:50.10	7.2	27.3
3	4:35.2	10.1	33.7	6:01.0	14.37	44.2

4.2.1 Descrição Sumária dos Testes Efetuados

Teste 1 : *Loop* com 50 iterações executando um comando de impressão do conteúdo do contador e um comando simples.

Teste 2 : Teste 1 acrescido de testes condicionais e execução de comandos com expansão de nome de arquivo.

Teste 3 : Teste 2 acrescido de substituição de comando, redirecionamentos e execução condicional.

Obs. : Não foi possível comparar o tempo na execução de expressões aritméticas pois a *ish* não possui este recurso internamente.

4.2.2 Análise dos Resultados

Através destes testes e de outros medidos durante a fase de teste do compilador foram tiradas as seguintes conclusões :

- Programas pequenos : Nestes programas o ganho (atual) na compilação é pequeno. Isto é devido principalmente ao *overhead* imposto pela fase de iniciação das tabelas do compilador (tabela de alias, variáveis e *path* de arquivos).
- Programas Maiores : Nestes programas o ganho é significativo, o tempo gasto na iniciação influencia muito pouco na soma total dos tempos. Para estes programas a compilação é fortemente recomendada, pois além das vantagens da compilação apresentadas no início deste capítulo temos um ganho efetivo em tempo de execução.

Infelizmente não foram possíveis comparações mais profundas devido à falta de uma *shell* no sistema com as mesmas características da *CCSH*.

Acreditamos que várias otimizações ainda podem ser efetuadas no compilador, dentre estas podemos citar uma melhor utilização dos recursos da linguagem C (em muitos pontos do código a eficiência foi substituída pela clareza). Esta primeira versão do compilador serviu principalmente para provar a viabilidade do compilador de linguagem de comando e para a definição de esquemas de execução para os principais comandos da linguagem escolhida como protótipo.

4.3 Extensões

A seguir serão apresentadas algumas sugestões para a continuidade deste trabalho. Algumas destas sugestões já estão sendo implementadas e provavelmente constarão da próxima versão do compilador.

- Implementação de uma versão interpretada e outra compilada para o *IBM-PC*. Devido ao fato desta máquina ter se tornado um equipamento bastante difundido e por possuir uma linguagem de comando sem muitos recursos, seria de interesse a implementação destes programas. No apêndice B serão mostradas algumas sugestões de como isto pode ser feito.
- Expansão dos recursos da linguagem implementada. Além dos comandos **read var** e **print expr** implementados nesta versão, outros comandos poderiam ser incluídos de modo a facilitar o interfaceamento com o usuário.
- Aprimoramento das estruturas de dados da biblioteca. Como a ambição inicial era o estudo da viabilidade da implementação de compiladores para linguagens de comando e a definição dos limites do que era compilável e o do que não era, não houve de início uma grande preocupação na "eficiência" das estruturas empregadas. Da primeira versão até a existente hoje, várias otimizações já foram feitas, mas acreditamos que ainda restam algumas, cuja implementação seria interessante.
- Implementação de uma versão simplificada (ou restrita). Uma série de otimizações locais seriam interessantes para o aumento da eficiência na

execução do código gerado. A seguir serão apresentadas duas simplificações que podem ser implementadas de modo simples.

Para cada comando a ser executado é necessário uma consulta nas tabelas de *hashing* para a obtenção do seu *pathname*. Na versão simplificada o usuário se encarregaria de ou passar ao compilador o *pathname* completo dos arquivos que deseja executar ou calculá-lo a partir da variável PATH do sistema em tempo de compilação.

Tomando cuidado na programação é possível retirar as transformações e verificações de tipo. Isto reduziria em muito o código da biblioteca.

- Implementar um interpretador para a C-SHELL no DIGIX [DIG87] e sistemas similares que ainda não possuam uma linguagem de comando de alto nível.
- Implementar um comando do estilo do *inline* de algumas linguagens. Através deste comando seria possível a inserção direta de fragmentos de código em linguagem "C" no programa gerado.

Apêndice A

Exemplos de Código Gerado Pelo Compilador

```
#
#          uso do compilador CSH : primeiro exemplo
#

# execucao de comandos simples com expansao de nome de arquivo

cat *.c > tudo.c

# execucao de pipeline de comando com redirecionamento de entrada
# no primeiro comando e saida no segundo

ccsh < teste.csh | cb > nome.c

# execucao condicional de comando

comando1 && comando2 || comando3

# uso de alias de comando

alias cc cc -c -O

# uso de variaveis

set files = ( *.c *.h)
echo $files a serem compiladas
```

```

# uso de expressoes numericas e comando while
set i = 1
while( $i < 10)
    echo "aguarde mais um pouco"
    @i++
end

```

Código Gerado :

```

#include "/usr/include/ccsh/var.h"
#include "/usr/include/ccsh/ccsh.h"
#include "/usr/include/ccsh/var.e"
#include "/usr/include/ccsh/expr.e"
#include "/usr/include/ccsh/path.e"
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i,j,k,fdi,fdo,fp[2],cnt,n_arg,stat=0;
    char buffer[MAXARGS][MAXTAMARG];
    char path[64],*s,*ccstr;
    inicvar(argc,argv);
    create_tab();
    cria_tabela_alias();
    for(i=0;i<MAXARGS;i++) arg[i]=buffer[i];

    iarg=0;
    strcpy(arg[iarg++],"cat");
    expande_n_arquivo(".", "*.c",arg,&iarg);
    executa(0,1,0,0,0,"","tudo.c");

    iarg=0;
    strcpy(arg[iarg++],"ccsh");
    executa(1,0,0,1,0,"teste.csh","");
    iarg=0;
    strcpy(arg[iarg++],"cb");
    executa(0,1,1,0,0,"","nome.c");
    iarg=0;

```

```

strcpy(arg[iarg++], "comando1");
executa(0,0,0,0,0, "", "");
if(!stat) goto lb2;
iarg=0;
strcpy(arg[iarg++], "comando2");
executa(0,0,0,0,0, "", "");
if(stat) goto lb2;
iarg=0;
strcpy(arg[iarg++], "comando3");
executa(0,0,0,0,0, "", "");

lb2:

define_alias("cc", "cc -c -O");

push_id("files", INSERT);
inicio_dcl_vetor();
iarg=0;
expande_n_arquivo(".", "*.c", arg, &iarg);
ins_vet(arg, iarg, 's');
iarg=0;
expande_n_arquivo(".", "*.h", arg, &iarg);
ins_vet(arg, iarg, 's');
dcl_vetor('s');

iarg=0;
strcpy(arg[iarg++], "echo");
push_id("files", INSERT);
push_valor();
subs_var(arg, &iarg);
strcpy(arg[iarg++], "a");
strcpy(arg[iarg++], "serem");
strcpy(arg[iarg++], "compiladas");
executa(0,0,0,0,0, "", "");

push_id("i", INSERT);
car_int(1);
atribuicao('s');
while(1){
    push_id("i", INSERT);
    push_valor();
    car_int(10);
    menor();
    if (!retira_topo()) break ;
}

```

```
    iarg=0;
    strcpy(arg[iarg++], "echo");
    strcpy(arg[iarg++], "aguarde mais um pouco");
    executa(0,0,0,0,0, "", "");
    push_id("i", INSERT);
    incrementa_1();
}
}
```

Apêndice B

Descrição Sucinta dos Módulos do CCSH

B.1 Introdução

Os programas fontes que compõem o compilador foram implementados e documentados de forma que qualquer usuário, com um conhecimento razoável de compiladores e de programação a nível de chamada de primitivas do sistema, possa modificá-lo de acordo com a sua necessidade. É nossa intenção que esse trabalho tenha continuidade em sistemas que não possuam uma linguagem de comando com recursos equivalentes aos mostrados pela C-SHELL.

A partir do fonte do compilador é possível, apesar de não ser trivial, gerar uma versão interpretada para a C-SHELL. Isto é interessante para sistema que ainda não dispõem de um interpretador de comandos desse nível (*e.g.* DIGIX). Neste caso as modificações serão realizadas quase que exclusivamente no fonte para o *yacc*, o código da biblioteca modifica muito pouco.

Foi assim que no primeiro semestre de 1988 dois alunos do curso de MM-986 implementaram um interpretador C-SHELL para o DOS. O interpretador implementado é na verdade um subconjunto significativo da C-SHELL e foi testado, embora não exaustivamente, no laboratório de *software* do DCC-UNICAMP.

B.2 Descrição dos Módulos que Compõem o Compilador

O compilador está dividido em duas partes : Uma que trata da compilação de programas escritos em C-SHELL e uma outra que trata da biblioteca do sistema. A seguir serão apresentados os módulos que compõem cada uma das partes do compilador.

B.2.1 Compilador

Módulo *gram.y*

Contém a gramática, no formato adequado ao *yacc*, e rotinas auxiliares para geração de código para alguns comandos.

Módulo *lex.l*

Conjunto de expressões regulares que formam o fonte para o *lex* e rotinas auxiliares para a análise léxica.

Módulo *ccsh.c*

Programa principal e rotinas para tratamento das opções do compilador.

B.2.2 Biblioteca

Módulo *executa.c*

Responsável pela execução de comandos externos (aplicativos).

Funções Principais :

executa(redin,redout,pipe_a,pipe_d,back,fnin,fnout) : Executa um comando de acordo com os parâmetros:

redin : indica a presença/ausência de redirecionamento de entrada;

redout: indica a presença/ausência de redirecionamento de saída;

`pipe_a` : indica a presença/ausência de *pipe* antes do comando; a entrada do comando tem que ser redirecionada para o *pipe*.

`pipe_d` : indica a presença/ausência de *pipe* depois do comando; a saída do comando é redirecionada para o *pipe*.

`back` : indica se o comando será executado no *background* ou não.

`fnin` e `fnout` : são os nomes dos arquivos que serão redirecionados para a entrada e saída, respectivamente.

`ver_redout()` : verifica se pode ocorrer redirecionamento de saída.

Módulo `expr.c`

Tratamento de expressões. Os operandos das expressões, quando não estiverem sendo explicitamente passados como parâmetros, estarão em *pilha[topo]* e em *pilha[topo-1]* .

Funções principais :

`atribuição(tipo)` : trata o comando de atribuição de inteiro ou *string* de acordo com *tipo*.

`push_id(nome,modo)` : insere/pesquisa variáveis na tabela de símbolos.

`push_valor()` : carrega na pilha o valor de uma variável

Expressões Relacionais e Lógicas : A função de cada uma das rotinas apresentadas a seguir é indicada pelo seu próprio nome.

`aval_or()`

`aval_and()`

`ou_logico()`

`ou_exclusivo_logico()`

`e_logico()`

`igualdade()`

`desigualdade()`

`menor_igual()`

`maior_igual()`

memor()
maior()
nega()
complementa()

Expressões de Rotação :

desl_direita()
desl_esquerda()

Expressões Aritméticas :

soma()
subtrai()
multiplica()
divide()
modulo()

Expressões de Incremento/Decremento :

incrementa_1()
decrementa_1()

Expressões de Atribuição :

sub_atr()
soma_atr()
mult_atr()
div_atr()

mod_atr()

Outras :

exp_reg(padrao, str) : avalia se *str* “casa” com *padrao*.

file_expr() : avalia expressões sobre arquivos.

Módulo **var.c**

Contém as rotinas para tratamento de variáveis.

Funções principais :

set_var() : marca uma variável como definida.

teata_var() : verifica se uma variável está ligada ou não.

unset_var() : desliga uma variável.

ins_vet(arg, iarg, tipo) : insere elemento nos vetores.

leia_var(varlist) : lê valores para as variáveis de *varlist*.

print_var(varlist) : imprime os valores das variáveis de *varlist*.

set_status(status) : atualiza o valor da variável interna *status*.

set_child(pid) : atualiza o valor da variável interna *pid*.

n_de_elementos(var) : retorna o número de elementos de uma variável.

inic_var() : insere as variáveis internas na tabela de símbolos.

al_chunk() : aloca um *chunk* de memória.

free_chunk() : libera um *chunk* de memória.

acess_elemento(j) : empilha o valor do elemento *j* do vetor.

inicio_dcl_vetor() : aloca o primeiro *chunk* de memória e inicia apontadores.

fecha_declaracao() : fecha a definição de vetor.

shift(i) : desloca o vetor *i* posições para a direita.

unshift(i) : desloca o vetor *i* posições para a esquerda.

Módulo `subs.c`

Executa as diversas substituições da C-SHELL.

Funções principais :

`define_alias(str1,str2)` : cria o sinônimo *str1* para a *string str2*.

`sub_alias(str1)` : devolve o sinônimo para *str1*.

`expande_n_arquivo(dir,padrao,arg,iarg)` : expande *padrao* em *dir* carregando os valores em *arg*.

`subs_var(arg,iarg)` : expande o conteúdo de variáveis no vetor de argumentos.

`subs_comando(n_arg,iarg,arg)` : trata substituição de comando.

Módulo `path.c`

Controla a tabela de *hash*¹ de *pathnames* de arquivos e os comandos que alteram o diretório corrente.

Funções principais :

`creat_tab()` : cria a tabela de *hash* a partir dos valores da variável `PATH`.

`insert_tab(arq,dir)` : insere o par *arq* e *dir* na tabela.

`pesq_tab(path,arq)` : pesquisa na tabela pelo *path* de *arq*.

`rehash()` : refaz a tabela de *hash*.

`cd_home()` : transfere o diretório corrente para o *home directory*.

`cd_vdir(var)` : transfere o diretório corrente para o diretório indicado pela variável *var*.

`cd_ndir(str)` : transfere o diretório corrente para o valor de *str*.

¹as tabelas de *hashing* foram implementadas segundo o algoritmo D da seção 6.4 de [KNU73]

Módulo `hash.c`

Controla a tabela de *hash* para *alias*.

Funções principais:

`hcreate(n)` : cria uma tabela com *n* elementos.

`hdestroy(n)` : destroi uma tabela com *n* elementos.

`hinsert(item)` : insere *item* na tabela de *hashing*.

`hsearch(item,acao)` : pesquisa pela chave na tabela de *hashing*.

Módulo `parserlc.c`

Rotinas para análise da linha de comando em busca de substituições que foram geradas pela execução de alguma outra substituição.

Devido à existência de uma série de substituições na linha de comando é possível que algumas destas substituições gerem outras substituições (ex. uma substituição de variável pode ocasionar uma expansão de nome de arquivo). Daí a necessidade de se fazer uma nova análise da linha de comando antes da execução do comando em busca de novas substituições. A verificação obedece a ordem das substituições conforme será apresentado no manual do usuário (apêndice c).

Funções principais :

`ver_alias()` : pesquisa *alias* na linha de comando.

`ver_variaveis()` : pesquisa substituição de variáveis na linha de comando.

`ver_expansao()` : pesquisa expansão de arquivo na linha de comando.

`parserlc()` : controla as rotinas anteriores.

B.3 Instalação do Compilador e da Biblioteca

A instalação do compilador é processada automaticamente pelo programa *make* [FEL78]. A seguir serão listados a base dos *makefiles* (arquivos fontes para o *make*). A versão completa do *makefile* é mais extensa e considera a instalação do compilador em sistemas diferentes do DIGIX.

B.3.1 Construção do Compilador

```
#
#      Makefile para construcao do compilador
#
#              v 1.0 (DIGIX)
#

CCFLAGS = -c -O
LDFLAGS =

#      arquivos que contem os fontes do compilador

FONTES = gram.c lex.c display.c gera.c ccsh.c yyrots.c

OBJETOS = gram.o lex.o display.o gera.o ccsh.o yyrots.o

ccsh : ${OBJETOS}
      cc ${LDFLAGS} ${OBJETOS} -o ccsh

#      construcao do analisador sintatico

gram.c : gram.y
      yacc -d gram.y
      mv y.tab.c gram.c

#      construcao do analisador lexico
lex.c : lex.l lex.h
      lex lex.l
      mv lex.yy.c lex.c

lex.h : gram.y
      mv y.tab.h lex.h

#      compilacao dos programas fontes

.c.o :
      cc ${CCFLAGS} $<
```

B.3.2 Construção da Biblioteca

```
#
#      Makefile para construcao da biblioteca do compilador
#
#              v 1.0 (DIGIX)
#

CCFLAGS = -c -O
LDFLAGS =

FONTES = path.c expr.c var.c subs.c parserlc.c executa.c
OBJETOS= path.o expr.o var.o subs.o parserlc.o executa.o

libccsh.a : ${OBJETOS}
    rm -f libccsh.a
    ar -r libccsh.a ${OBJETOS}
    ranlib libccsh.a

.o.c:
    cc -c $<
```

Apêndice C

Manual do Compilador CCSH

O manual foi composto no mesmo formato dos manuais existentes para o sistema *UNIX*. Para não fugir do padrão deixaremos o resto desta página em branco.

CCSH

NOME

CCSH – Compilador para a linguagem de comando C-SHELL

SINOPSE

```
ccsh [- opcoes ] [ arg ..... ]
```

DESCRIÇÃO

CCSH é um compilador para a linguagem de comando C-SHELL do BSD Unix. Dentre as principais características desta linguagem se destacam a sua sintaxe *c-like*, existência de variáveis, expressões e *alias* (sinônimo para comandos).

O compilador CCSH gera código em linguagem “C” padrão, para posterior compilação, e ligação com a biblioteca *lccsh.a*.

ESTRUTURA LÉXICA

Os caracteres *branco* e *tab* são os separadores de palavras reconhecidos pelo compilador; os caracteres '&' '|' ';' '<' '>' '(' ')' apesar de serem considerados separadores, representam símbolos especiais para o compilador; os caracteres '||' '&&' '>>' formam um único símbolo; o caracter '\ ' inibe a interpretação primeiro caracter. É conhecido como caracter de escape e é usado para inserir os caracteres especiais (metacaracteres) em palavras.

Strings são delimitadas pelos caracteres ' ' e ' " '. Caracteres especiais contidos na *string* não formam palavras separadas. Para continuar uma *string* em uma outra linha é necessário utilizar o caracter de escape. O caracter '# ' inicia um comentário que continua até o final da linha.

COMANDOS

Um comando simples é uma seqüência de palavras, sendo que a primeira delas é o comando a ser executado. Um comando ou uma seqüência de comandos simples separados pelo caracter '|' formam um *pipeline*. A saída de um comando num *pipeline* é conectada à entrada do próximo comando. Seqüências de *pipelines* podem ser separadas pelo caracter ';', e são executados seqüencialmente. Uma seqüência de

pipeline pode ser executada de forma que a próxima seqüência necessite esperar por seu término, basta anexar o caracter '&' ao fim da seqüência.

Qualquer uma das formas anteriores podem ser delimitadas pelos caracteres '(' e ')' para formar um comando simples, que por sua vez pode ser um coponente de um *pipeline*. É também possível separar *pipelines* com os símbolos '||' e '&&' indicando, como na linguagem "C", que o segundo componente será executado se e somente se o primeiro falhar ou suceder respectivamente.

SUBSTITUIÇÕES

Serão descritas a seguir as várias transformações que a *shell* executa na linha de comando antes de sua execução. As substituições serão apresentadas na mesma ordem em que elas ocorrem.

Nota: *Strings* delimitadas pelo caracter '`' não estão sujeitas a substituições, enquanto que as *strings* delimitadas pelo caracter '"' estão sujeitas à substituição de variáveis e comando.

Alias

A *shell* mantém uma lista de sinônimos que podem ser criados, modificados e vistos pelos comandos *alias* e *unalias*. Após ser lida a linha de comando ela é analisada e separada em comandos; a primeira palavra da esquerda para a direita de cada comando é pesquisada na lista de sinônimos, se constar da lista de sinônimos será substituída pelo texto presente nesta lista.

Assim se o sinônimo para *l* é *ls -ali* o comando *ls /usr* é substituído por: *ls -ali /usr*; a lista de argumentos permanece inalterada.

Substituição de Variável

A *shell* mantém um conjunto de variáveis, cada uma das quais possuindo uma lista de zero ou mais palavras. Algumas destas variáveis são criadas pela *shell* ou utilizadas internamente por ela. Como exemplo, pode-se citar a variável *argv*, ela é uma cópia da lista de argumentos da *shell*, o seu valor pode ser referenciado de um modo especial, como será visto a seguir.

Os valores das variáveis podem ser vistos e modificados utilizando os comandos *set* e *unset*. Das variáveis referenciadas pela *shell* algumas atuam como chaves, o seu valor não tem significado para a *shell*. O que interessa é se elas foram referenciadas pelo comando *set* ou não (se estão "ligadas" ou "desligadas"). A variável *verbose*, por exemplo, quando ligada causa a impressão da linha de comando antes da execução. A opção *-v* da *shell* tem o mesmo significado de *set verbose*.

Algumas operações tratam o valor das variáveis como números. O comando '@', permite o cálculo de expressões numéricas e atribui o valor da expressão para a variável. Na C-SHELL os valores das variáveis são sempre tratados como *strings*, na implementação deste compilador houve uma diferenciação entre valores numéricos e alfanuméricos, apesar disto as conversões entre os tipos são permitidas desde que obedecem às regras de formação dos números.

Após a substituição *alias* e após a análise da linha de comando, e antes da execução do comando, é executada a substituição de variáveis. Para tal é pesquisada a existência do caracter '\$' precedendo um nome de variável na linha de comando. O nome do comando e a lista de argumentos são expandidos ao mesmo tempo. Se a primeira palavra da lista for uma expansão de variável, a primeira palavra gerada por esta expansão é o comando a ser executado.

Formas de uso das variáveis na *shell*:

\$nome
\${nome}

São substituídas pelo valor (palavras) da variável *nome*, cada uma delas separadas por um branco. O uso de chaves permite que seja anexado um texto ao conteúdo de uma variável (ex. suponha que o conteúdo da variável *\$prefix* seja : CLAUDIO, então *\$prefix{MAR}* será substituído por CALUDIOMAR).

\$nome[seletor]
\${nome}[seletor]

Pode ser utilizado para selecionar algumas das palavras da variável *nome*. O seletor pode ser uma variável, um número ou um limite (dois números separados pelo caracter '-', ex. 1-5). O seletor '*' indica todos os elementos da variável.

##nome
##{nome}

Número de palavras de uma variável.

\$n
\${n}

Estas duas construções são equivalentes a *argv*[*n*].

\$*

Equivalente a *\$argv*{*}

\$?nome

\${nome}

Substitui a *string* '1' se *nome* estiver ligada, e '0' caso contrário.

\$\$

Substitui o número do processo pai da *shell* (em decimal).

Obs : os modificadores de variáveis encontradas na C-SHELL não foram implementados nesta versão.

Substituição de Comando e de Nome de Arquivo

Substituição de Comando

Esta substituição é indicada por comandos delimitados pelo caracter ` `'. A saída destes comandos substituem o texto original na linha de comando. Strings nulas produzidas na saída do comando são descartadas, *brancos*, *tabs* e *newlines* são considerados separadores de palavras.

Expansão de Nome de Arquivo

Se uma palavra contém algum dos caracteres '*', '?', '[' ou '{' ou começa com o caracter '~', está sujeita à expansão de nome de arquivo. Esta palavra é vista como um padrão e é substituída pela lista ordenada de nomes de arquivo que casam com este padrão. É considerado erro a especificação de um padrão que não casa com nenhum nome de arquivo no diretório especificado.

Na expansão de nome de arquivo, o caracter '.' no início de um nome, ou imediatamente após o caracter '/', bem como este caracter, devem ser casados explicitamente. O caracter '*' casa qualquer *string* de caracteres, inclusive a *string* nula. O caracter '?' casa um único caracter. A seqüência '[...]' casa qualquer um dos caracteres delimitados. Um par de caracteres separados por '-' dentro de '[...]' casa qualquer caracter lexicamente entre eles.

O caracter '~' no início de um nome de arquivo é utilizado para referenciar o diretório base (*home directory*). Este caracter só formando

uma palavra, expande o valor da variável do sistema *home*.

REDIRECIONAMENTO DE ENTRADA E SAÍDA

A entrada padrão e a saída padrão de um comando podem ser redirecionadas de acordo com a sintaxe:

< nome

Abre o arquivo *nome* como entrada padrão.

> nome

>! nome

>& nome

>!& nome

O arquivo *nome* é utilizado como saída padrão. Se este arquivo não existir ele é criado; se ele existir o seu conteúdo será perdido. Se a variável *noclobber* estiver ligada e se o arquivo existir será emitida uma mensagem de erro. Este mecanismo evita que arquivos sejam destruídos acidentalmente. O caracter '!' pode ser utilizado para eliminar esta verificação. O caracter '&' redireciona a saída de erro padrão para o mesmo arquivo da saída padrão.

>> nome

>>& nome

>>! nome

>>!& nome

Utiliza o arquivo *nome* como saída padrão como no uso do caracter '>', concatenando a saída do comando no fim do arquivo. Se a variável *noclobber* estiver ligada, será considerado erro a não existência do arquivo. O caráter '!' quando usado evita este tipo de erro.

Um comando recebe o ambiente da *shell* acrescido das modificações impostas pelos vários tipos de redirecionamento e pela presença de comandos em um *pipeline*.

EXPRESSÕES

Um número grande de comandos internos implementam expressões, com operadores similares aos encontrados na linguagem "C", possuindo a mesma precedência. Estas expressões aparecem nos comandos

set, "@" (comando *set* para valores numéricos), *exit*, *if* e *while*. Os seguintes operadores estão disponíveis para o cálculo de expressões :

||, &&, |, ^, &, ==, !=, =~, !~, >=, <=, >, <, <<, >>, ++, --, + =, - =, * =, / =, %=, ^=, &=, +, -, *, /, %, !, ()

Aqui a precedência cresce para a direita, '=' '!=' ' =~ ' !~ ' <=' '>=' '<' e '>', '<<' e '>>', '+ ' e '- ', '* ' / ' e '%' estão, em grupos, no mesmo nível. Os operadores ==, !=, =~, !~ comparam seus argumentos como *strings*; todos os outros operadores tratam seus argumentos como números. Os operadores =~ e !~ são semelhantes aos operadores != e ==, exceto pelo fato de que seu lado direito é tratado como um padrão. Isto reduz a necessidade do comando *switch* nos programas escritos em *shell*, quando tudo que é realmente necessário é um casamento de padrão.

A C-SHELL possui ainda expressões da forma *-k nomearg*, onde *k* é uma das seguintes *flags* :

r	permissão para leitura
w	permissão para escrita
x	permissão para execução
e	existência
o	propriedade do arquivo
z	tamanho nulo
f	arquivo comum
d	diretorio

nomearg é testado para ver se possui a *flag* especificada para o usuário. Se o arquivo não existe ou se é inacessível então todas as *flags* retornam falso, i.é., 0.

COMANDOS INTERNOS

alias

alias nome

alias nome lista_de_palavras

A primeira forma imprime a tabela de *alias*. A segunda imprime o sinônimo existente para *nome*. A última insere na tabela de *alias* o sinônimo para *nome*.

break

Executa o primeiro comando após o *end* do comandos *while* ou *foreach* mais próximo.

breaksw

Forma semelhante ao *break* utilizado no comando *switch*. Desvia para o primeiro comando após *endsw*.

case label:

Parte do comando *switch* como será visto a seguir.

cd

cd nome

chdir

chdir nome

Muda o diretório de trabalho para o diretório *nome*. Se nenhum argumento for especificado o diretório de trabalho passa a ser o diretório base (*home directory*) do usuário.

continue

A semântica deste comando é idêntica à da linguagem "C". Continua a execução do comando *while* ou *foreach* mais interno, saltando os comandos até o *end*.

default:

Rotula a cláusula *default* de um comando *switch*.

else

end

endif

endsw

Veja a descrição dos comandos *foreach*, *if*, *switch*, e *while* a seguir.

exit

exit(expressão)

Termina a *shell* com o *status* igual ao da variável *status* (primeira forma) ou igual ao valor da expressão (segunda forma).

foreach in nome (lista_de_palavras)

.....

end

A variável *nome* recebe a cada iteração um dos valores presentes na *lista_de_palavras*.

goto palavra

Transfere o fluxo para a posição do programa rotulada por *palavra*. *Palavra* pode sofrer expansão de nome de arquivo e substituição de comando.

if (expr) comando_simples

Se *expr* resultar em um valor verdadeiro (não zero), então *comando_simples* será executado.

if (expr) then

.....

else if (expr2) then

.....

else

.....

endif

Se *expr* resultar num valor verdadeiro então os comandos até o primeiro *else* serão executados; senão se *expr2* resultar num valor verdadeiro, então os comandos até o segundo *else* serão executados, etc. Não existe limites para os pares *else-if*; somente um *endif* deve ser utilizado para terminar o comando *if*.

kill pid

kill -sinal pid

Envia o sinal TERM (sinal de término) ou o sinal especificado para o processo *pid*.

rehash

Faz com que seja remontada a tabela interna de *hash* de acordo com o conteúdo da variável *path*.

repeat cont comando

Repete o comando especificado *cont* vezes.

```
set
set nome
set nome=palavra
set nome[n]=palavra
set nome=(lista_de_palavra)
```

A primeira forma mostra o valor de todas as variáveis ligadas. A segunda forma atribui à variável *nome* a *string* vazia (liga a variável). A terceira forma atribui à variável *nome* o valor *palavra*. A quarta forma atribui ao *n-ésimo* elemento do vetor *nome* o valor *palavra*; este elemento deve existir. A última forma atribui a *nome* o vetor *list.de.palavras*.

setenv nome valor

Atribui *valor* à variável do sistema *nome*. As variáveis do ambiente mais comumente usadas (TERM, USER e PATH) são automaticamente importadas e exportadas pela *shell*, não é necessário utilizar o comando *setenv* para iniciar seu conteúdo.

source filename

equivalente a um *include* no arquivo *filename*.

switch (string)

```
case str1:
.....
breaksw
.....
default:
.....
breaksw
endsw
```

A semântica é idêntica ao comando de mesmo nome da linguagem "C". A *strings* das cláusulas *case* podem conter expressões regulares.

unalias nome

Retira da lista de alias a entrada correspondente a *nome*.

unset nome

Retira da tabela de símbolos a variável *nome*.

unsetenv nome

Idem.

while (expressão)

.....

end

Enquanto o valor da expressão for verdadeiro, os comandos subsequentes até o comando *end* serão executados.

@

@ nome= expressão

@ nome[n]=expressão

A primeira forma imprime o valor de todas as variáveis da *shell*. A segunda atribui à variável *nome* o valor calculado para a expressão. A terceira forma atribui ao *n-ésimo* elemento do vetor *nome* o valor calculado para a expressão.

Variáveis Predefinidas e Variáveis do Ambiente

As variáveis a seguir têm um significado especial para a *shell*. Destas, *argv*, *cwd*, *home*, *path* e *status* são sempre criadas pela *shell*. Excetuando *cwd* e *status* todas as iniciações são ocorridas durante a iniciação do sistema; estas variáveis não serão modificadas pela *shell*, elas podem, entretanto, ser modificadas pelos usuários.

A *shell* copia o valor da variável de ambiente *USER* na variável *user*, *TERM* na variável *term* e *HOME* na variável *home*. A variável *PATH* é igualmente tratada.

Variáveis e Respectivas Funções

argv :

Contém a lista de argumentos passados para a *shell*. \$n pode ser utilizado no lugar de *argv[n]*.

cwd

Contém o valor do *pathname* do diretório corrente.

echo

É ligada quando a *shell* é chamada com a opção *-x*. Quando ligada causa a impressão do comando antes de sua execução.

home

Contém o diretório base do usuário, iniciada a partir da variável do ambiente *HOME*. A expansão de nome de arquivo '~' expande este valor.

noclobber

Se ligada, como descrito na seção de redirecionamento de entrada e saída, são impostas restrições no redirecionamento de saída, protegendo os arquivos de destruição acidental.

noglob

Se ligada, expansão de nome de arquivo é inibida. Isto é interessante em programas que não tratam de nome de arquivos.

nonomatch

Se ligada, não é considerado erro se numa expansão de nome de arquivo não existir arquivos que casem o padrão especificado; em lugar disto o padrão será retornado.

path

Cada palavra da variável *path* especifica um diretório no qual um arquivo executável será pesquisado para execução. Se não existir esta variável somente serão executados os arquivos que contem o *pathname* completo.

status

Contém o *status* retornado pelo último comando executado.

Opções do Compilador

-n

Os comandos são analisados mas não são executados.

-x

Liga a variável *echo* da *shell*.

AUTOR

Luiz C. D. Carneiro, GDF - DCC - IMECC - UNICAMP

ARQUIVOS

/usr/lib/libccsh.a
include files

LIMITAÇÕES

Comandos presentes na CSH e ainda não implementados no compilador .

history, onintr label, inline data ou here document (fornecer dados a comandos) e flags de variáveis (ex.: *\$i:t*).

As constantes MAXNVAR, MAXNARQ, MAXARG, MAXENTRY, etc... definidas nos arquivos com extensão ".h" limitam o tamanho das tabelas internas da biblioteca. Para maiores detalhes veja os arquivos com esta extensão que acompanham o compilador.

VEJA

sh(1), csh(1), stat(2), exec(2), fork(2), pipe(2), wait(2)

BUGS

Nesta versão do compilador a execução será abortada assim que uma situação de erro for detectada. Em alguns pontos isto ainda não está sendo feito.

Esta é uma versão experimental, os erros detectados deverão ser comunicados ao autor para análise e correção.

Bibliografia

- [AHO74] AHO, A. V. ; HOPCROFT, J. E. ; ULLMAN, J. D. *The design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974, 796p.
- [AVR85] AVRITZER, A. *Implementação do Mshell, Linguagem de Comandos do Mzunix*, Anais do IV Congresso da SBC, Viçosa, MG, jul 1984, 363-374.
- [BAR72] BARRON, D. W. and JACKSON, I. R. *The Evolution of Job control Languages*, Software-Practice and Experience, Vol. 2 n. 2, Apr-Jun 1972, pp. 143-164.
- [BAR74] BARRON, D. W. *Job Control Language and Control Programs*, The Computer Journal, vol. 17 n. 3, Aug 1974, pp. 282-286.
- [BOU78] BOURNE, S. R. *An Introduction to the Unix Shell*, Bell Laboratories, Murray Hill, New Jersey, Nov 1978, 25pp.
- [BOU83] BOURNE, S. R. *The Unix System*, Addison-Wesley, 1983, 351 pp.
- [BRE84] BRESNAHAN, J. B. and BARNARD, D. T. and MacLEOD, I. A. *WSH - A New Command interpreter for Unix*, Software-Practice and Experience, vol. 14 n. 12, Dec 1984, pp. 1197-1205.
- [COW75] COWAN, R. M. *Burroughs 6700/7700 Work Flow Language*, em [UNG75].
- [DIG87] DIGIREDE *Sistema Operacional Digix*, Manual de Operação, Vol III, São Paulo, SP, fev 1987, 1-27.
- [DGT83] DIGITAL RESEARCH, *CP/M Plus Version 3, Programmers Guide*, Pacific Grove, 1983.

- [DRU87a] DRUMMOND,R. and LIESENBERG,H. K. E. *Projeto AIDS*, Comunicação interna, DCC-UNICAMP, 1987.
- [DRU87b] DRUMMOND,R. and LIESEMBERG,H. K. E. *A-HAND*, Comunicação interna, DCC-UNICAMP, 1987.
- [ELL80] ELLIS,J. R. *A Lisp Shell*, Sigplan Notices, Vol. 15 n. 5, May 1980, pp. 27-34.
- [FEL78] FELDMAN,S. I. *Make - A program for Maintaining Computer Programs*, Bell Labs. CSTR, 1978.
- [FRA83] FRASER,C. W. and HANSON,D. R. *A High-Level Programming Language*, Sigplan Notices, Vol. 18 n. 6, Jun 1983, pp. 212-219.
- [GRA75] GRAM,C. and HERTWECK,F. *Command Languages: Design Considerations an Basic Concepts*, in [UNG75].
- [HAR82] HARDY Jr.,I. T. *The Syntax of Interactive Command Languages: A Framework for Design*, Software-Practice and Experience, vol.12(1), Dec 1982, pp. 67-75
- [HOL83] HOLT,R. C. *Concurrent Euclid, The Unix System and Tunis*, Addison-Wesley, Reading, Massachusetts, 1983, 323 pp.
- [JOH78] JOHNSON,S. C. *YACC - Yet Another Compiler-Compiler*, Bell Labs. CSTR 332, 1978.
- [JOH80] JOHNSON,S. C. *Language Development Tools on the Unix System*, Computer, Aug 1980, pp. 16-23.
- [JOY80] JOY,W. *An Introduction to the C Shell*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, Nov 1980, 46 pp.
- [KER78a] KERNIGHAN,B. W. and RITCHIE,D. M. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978, 228 pp.
- [KER78b] KERNIGHAN,B. W. and RITCHIE,D. M. *Unix Programming*, Bell Labs.
- [KER81] KERNIGHAN,B. W. and MASHEY,R. J. *The Unix Programming Environment*, Computer, Apr 1981, pp. 12-24.

- [KER84] KERNIGHAN,B. W. ; PIKE,R. *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984, 357 pp.
- [KNU73] KNUTH,C. E. *The Art of Computer Programming, Vol. 3 : Searching and Sorting*, Addison Wesley, 1973.
- [LES75] LESK,M. E. *LEX - A Lexical Analyzer generator*, Bell Labs. CSTR 39, 1975.
- [LEV80] LEVINE,J. *Why a Lisp-Based Command Language ?*, Sigplan Notices, vol. 15 n. 5, May 1980, pp 49-53.
- [MAL81] MALCOLN,J. A. *Brevity and Clarity in Command Languages*,Sigplan Notices, vol. 16 n. 10, Oct 1981, pp. 53-59.
- [QUA85] QUARTERNAN,J. S. and SILBERSCHATZ,A. and PETERSON,J. L. *4.2 BSD and 4.3 BSD as Example of the Unix System*, Computing Surveys, vol. 17 n. 4, Dec 1985.
- [RIT78] RITCHIE,D. M. ; THOMPSON,K. *The Unix Time Sharing System*, The Bell System Technical Journal, 57(6), Jul-Aug 1978, pp. 1905-29.
- [RIT79] RITCHIE,D. M. *The Evolution of the Unix Time Sharing System*, Lecture Notes in Computer Science, vol 79, Sep 1979, pp. 23-35.
- [SCH85] SCHREINER, A. T. ; FRIEDMAN,H. G. *Introduction to Compiler Construction With Unix*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985, 194p.
- [SOB85] SOBELL,M. G. *A Practical Guide to UNIX System V*, The Benjamin/Cummings Publishing co., Menlo Park, California, 1985, 577 pp.
- [SED84] SEDGEWICK, R. *Algorithms*, Addison-Wesley Publishing Co., 1984, 552p.
- [THO78] THOMPSON,K. *Unix Implementation*, The Bell System Technical Journal, 57(6), Jul-Aug 1978, pp. 1931-46.
- [TOS85] TOSCANI, *Linguagem de Comando S-SHELL*, anais do V Congresso Brasileiro de Computação

- [UNI83] UNISOFT *Uniplus+ System V, User's Manual*, sections 1-6, Berkeley, California, 1983.
- [UNG75] UNGER,C. ,ed., *Proc. of the IFIP Working Conference on Command Languages*, North Holland, 1975.
- [WAD75] WADA,E. *On the Possibility of the Unification of Command and Programming Language*, em [UNG75].