by Peter Collinson, Hillside Systems

GARY DAVIS

Analyzing Syntax

I think that nearly all the programs I have written in my life have contained an element of text processing. Some programs take input from users in the form of interactive dialogue; others read data from files. It has always been a sensible approach on UNIX systems to store primary data as text so people can use general-purpose editors to access and maintain the files and utilize the rich UNIX tool set to search or otherwise manipulate the data.

Programmers who plan to take text input from a file or from the user's keyboard face similar sets of problems when designing what that text should be. The fundamental problem is that the program is performing what amounts to pattern recognition. In general, the program will see the text come in one character at a time, and it needs to understand the format, allowing it to pick out meaningful information from the data stream. We, as humans, can experience the problem when listening to others talking in an unfamiliar language. If the language is completely unknown to us, then we will hear a meaningless jumble of sounds. However, we only need a very small understanding of the language to break those sounds down into words that we might learn to understand.

The task, then, of any program that wants to read text is to take the characters it sees and change that text stream into something comprehensible.

Much of our understanding about how text recognition should be done is taken from the work of the pioneers in program language design. Compilers and interpreters for languages like FORTRAN or BASIC tended to recognize text using a character-by-character approach. For early BASIC, each line started with a number that was the line number, statements looked like this:

```
10 INPUT A
20 B = A + 1
30 PRINT B
```

So for each line, the interpreter starts by reading numeric characters until it finds a character that is not a number. It will turn the sequence of characters into a decimal number that is the line number. Scanning on, the next non-space character will be the first character of a command or the start of an arithmetic statement. Characters are read until it's clear what the statement is. Then, the arguments to the statement are processed. This is an example of "bottom-up" parsing. We take the characters that make up the input and attempt to collect them together, making a higher-level object.

Early computer scientists realized the character-by-character approach does not scale. It doesn't allow you to share code. Each statement in the language is understood by an individual part of the program that is coded separately to deal with each type of statement. The process is error-prone. No formal rules are being applied to the text, so each statement has to be examined to see if it fits in with the rules of the language.

It was realized that the fundamental pattern recognition problem was actually one of language recognition, that programming a computer was a form of language use. Computer designers began to look at the work of linguists. Natural languages all have grammar, a set of rules that are used to create the phrases or sentences. If a grammar was imposed on a language intended for computer input, then it would be possible to use the rules of the grammar to verify that input from the user was syntactically correct. In fact, parts of the early languages used formal grammar; they used the familiar infix algebraic notation to express mathematical statements. It was a matter of generalizing the approach to a whole language.

These grammars tend to be "top-down." We start the translation process looking for a "program" that comprises several varieties of "statement." For primitive BASIC, each statement is a "line-number" followed by a "command." One command is a "keyword" (`INPUT` in the example above) followed by a "variable list," and a variable list is a single "variable name" or a variable name followed by a comma and another variable list.

The specification of the Algol language was the first time a complete programming language was described using a formal grammar. The grammar was specified with the reference language called Backus-Naur Form (or BNF) that was used by the international group of people who developed Algol. I suspect that Algol is always thought of as a European language by many folks in North America. It shouldn't be forgotten that John Backus was the U.S. representative on the committee that established the language, and that the language was from the beginning intended to be an international language.

Later, BNF became the basis for automatic language specification systems, such as the `yacc` language, which is used to provide syntax analysis for the C language.

Algol and the notion of using formal grammars to specify languages taught us that the business of recognition should be done in several phases: First, you should attempt to make sense of the characters by analyzing the text into identifiable chunks known as tokens. Then, having built a representation of the structure of the text, that structure can be examined using a top-down approach to check its syntax. Finally, having validated the input, the statement can be interpreted, compiled or stored, depending on the application.

Tokens and Shells

Well, this article started from the thought that I take the notion of a token for granted. When I am looking at C, a shell script, an `awk` script or Perl program, I mentally parse the line into tokens in the same way that you are reading this text. You know that each word is delimited by spaces and your braingetsveryconfusedwhentheyareomitted. Well, it's not quite true to say that words in English text are only delimited by spaces. Some words are terminated by punctuation characters. Punctuation is also used to give you information about the intended phrasing of the text. Also, some words are terminated by the end of a line.

Many UNIX programs use the approach of breaking the input line into tokens. Some programs will then use a formal grammar to process the tokenized data, often making use of `yacc` to generate the necessary tables. Many programs will stop at the tokenization phase. Shells are a good example of programs that split the input line into tokens and then use those tokens in a bottom-up fashion to execute a command for the user.

On the whole, throughout UNIX, there's a consistent view about what a token in a shell might be. Tokens on a shell input line are separated by spaces or tabs (which I tend lump together as "white space," although of course the color of the space depends on the settings on your screen). Shells will read their input and split the line into separate tokens, discarding any white space. As I said, having obtained a set of tokens, most shells don't possess a grammar that is formally expressible and will tend to analyze their input, doing what is needed depending on what they find.

All shells take the first token on the line to be a command name and will look for a file of that name in various well-known places defined by the `PATH` environment variable (or the `path` variable in `csh`). Other tokens on the line become arguments that are passed into the program as separate arguments. It's the job of the program to decode these arguments and take the action that the user has started.

The early UNIX shells did very little more than take input from the user and call commands. The shell did recognize that a token containing an asterisk or a question mark meant that the token should be expanded into a list of file names before the command is called, each of the matching file names becoming a separate argument to the program that is being invoked.

Later, variables were introduced into shells. When a token starts with a dollar character, it is assumed to be a variable that contains a value. When the command is being created, any contents of shell variables are inserted into the command at the appropriate position. So, for the Bourne shell,

```
$ src=fromfile
$ dst=tofile
$ cp $src $dst
```

The first statements set the variables `src` and `dst`. When the `cp` command is read, the `$src` and `$dst` tokens are replaced by the contents of the variables before the command is called.

Because the first object on a line is also a token, there is no reason why it cannot also be a shell variable, if we follow on from the example above and use `mv` as the command name rather than `cp`:

```
$ CPCMD=mv
$ $CPCMD $src $dst
```

Once you introduce characters that have a special meaning like dollar, asterisk or question mark, then you need a way of telling the shell not to use that special meaning, that you want the special character to be passed intact into the program that is being called. Actually, we also need a way of passing space or tab characters into the program. We need a method of telling the shell that the space should not act as a token delimiter but should be considered part of the token. We need a way to quote character sequences.

Quoting in shells is somewhat of a minefield. Each shell does things slightly differently. I will concentrate on the Bourne

shell (`sh`) and its offspring the Korn shell (`ksh`). Beware that the rules may be different if you are using publicly available clones of these shells. Also, quoting in `csh` is much more restricted than in the Bourne shell, making the Bourne shell the first choice for writing scripts.

In the Bourne shell, you can stop the special meaning of any single character by preceding it with a backslash. So,

```
$ echo hello world
hello world
```

will call the `echo` command with two arguments: `hello` and `world`, while

```
$ echo hello\ world
hello world
```

will call the command with a single argument `hello world`. However, using the backslash isn't particularly user-friendly; it looks kind of ugly. We usually use a pair of quote characters to create the single token:

```
$ echo 'hello world'
hello world
```

The shell sees the opening single quote and takes all the input that follows as part of the token until the closing quote is found. The quotes are discarded when the command is called. The Bourne shell also allows new lines to be enclosed in single quotes so single arguments that span several lines can be specified simply:

```
$ echo 'hello
world'
hello
world
```

The single quoted token is "super-quoted"; that is, it ignores all special characters, including the backslash. The only character that you cannot put inside a single-quoted token is another single quote. All other characters are carried through unchanged to the command that you are executing. Actually, missing an end quote in a complex shell script is a great way to introduce hard-to-find errors, so great care needs to be taken to match opening and closing quotes when laying out the text.

There are many occasions where you want to use the shell to create a single argument but also have part of the token created from shell variables. A double-quoted token achieves the effect:

```
$ src='hello world'
$ echo "$src and thanks"
hello world and thanks
```

Here, we set a variable to a value containing a space, then type an `echo` command. The `$src` is replaced in the token by

`hello world`, but because the string is quoted it will be passed as a single argument into the `echo` command. Note that we can do string formatting by adding text inside the quoted string.

A good and little-used feature of the token recognition system is that when we place two quoted sections together, they are treated as a single token. In shell scripts, I often use this feature to hop in and out of the different quoting forms:

```
$ echo 'It'"'"'s a small world after all'
It's a small world after all
```

adding a single quote into token that is passed into the `echo` command. I agree that the example above is overkill. It's hard to think of a real one that doesn't need 17 pages of explanation.

It's important to understand the Bourne shell, its quoting rules and variable expansion, because many other programs follow its lead. A good grasp of how things work in the shell helps considerably with programs like `awk` or Perl. The shell is somewhat odd because it's very much a complex string processing language sitting behind a language that implements a set of imperative commands that act on named files and have a set number of fixed parameters.

Tokens in Email

The use of formal grammars and tokens spring up in what you might think are odd areas. For example, we have implicit token recognition and syntax analysis when we send or receive email. Several fields in the mail header, including the sender address (the `From:` line) and the destination address (the `To:` line), are constrained to a certain grammatical form by RFC 822, which defines the way we all deal with mail on the Internet. RFC 822 also specifies the syntax of the `Date:` field, ensuring that the line is comprehensible to automatic systems that may wish to use it, perhaps for sorting the contents of your mailbox into chronological order.

Mail addresses are something that we use every day, and it's often poorly understood that there is a set of rules that govern their construction. Incidentally, what follows is a little abbreviated. You are encouraged to read RFC 822 to obtain the full gory unexpurgated details.

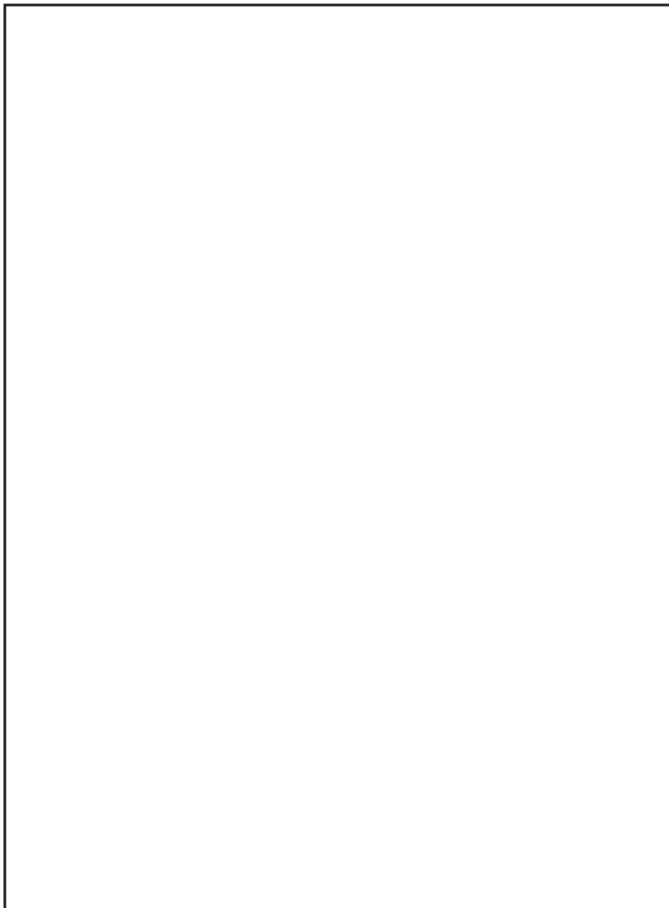
The simplest form of an address field is something like this:

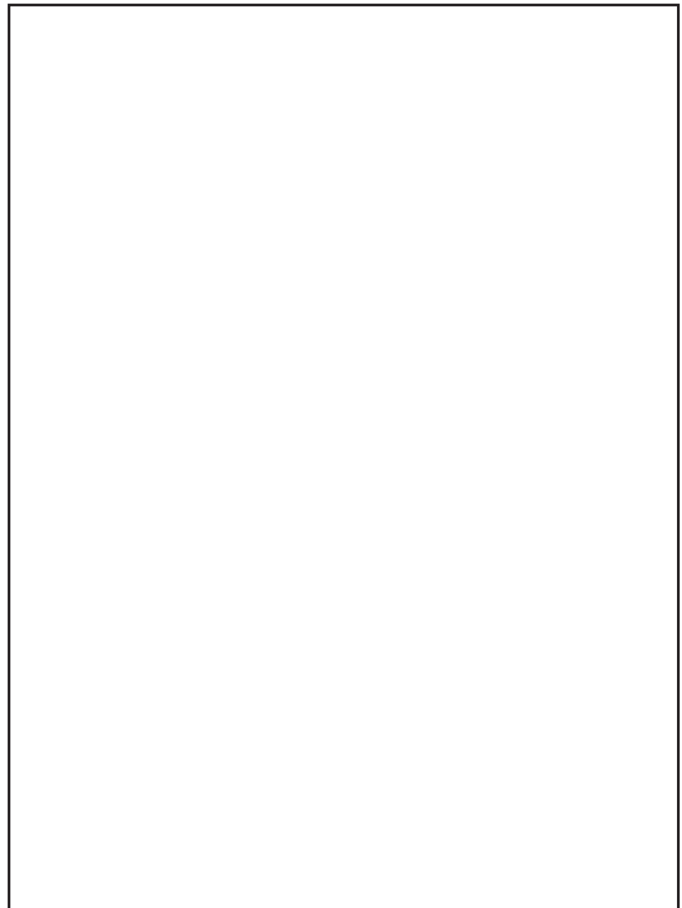
```
To: aperson@someplace.domain
```

As you know, you can place comma-separated lists of addresses on the line. Also the header line can be folded, by inserting a new line and at least one white space character at the start of the next line:

```
To: aperson@someplace.domain,
    another@somewhere.else
```

Notice that the line is separated into tokens using white space as the separator. This means that someone's mailbox address





cannot contain a space. Well, strictly it can, if you are prepared to always quote the name using double quotes:

```
To: "A Person"@someplace.domain
```

Actually,

```
From: A Person@someplace.domain
```

is syntactically incorrect. The use of double quotes to quote parts of the mail address is poorly understood, so you are creating grief for yourself if you create a mailbox address with a space. Incidentally, you can also use the backslash to quote a single character in addresses, but the standard explicitly says that

```
From: A\ Person@someplace.domain
```

is incorrect.

Most people like to put a human-readable name into the address, and the most common way is to use angle brackets:

```
From: Agnes Person <aperson@someplace.domain>
```

There are rules about what may be placed in the part of the address before the angle-bracketed section. You are not supposed to put one of the special characters used in the mail address here. So, for example, a period is not allowed unless you quote it, so

```
From: "A. Person" <aperson@someplace.domain>
```

is legal because it's a quoted string. Most mail systems don't enforce these particular rules, so there's considerable laxity in general practice.

The other common way of inserting your name into the address is to use the comment facility of the syntax. Any character can be placed inside round brackets and will be treated as a comment, ignored by the mail system:

```
From: aperson@someplace.domain (A. Person)
```

or

```
From: (A. Person) aperson@someplace.domain
```

Notice that the angle brackets have been removed. Strictly,

```
From: (A. Person) <aperson@someplace.domain>
```

is not legal. The commented section in round brackets acts like a single space, and so the line becomes

```
To: <aperson@someplace.domain>
```

which is not actually allowed in the syntax. Again, most mailers don't complain and will get on with the job of sending the mail.

As I said, there's a bunch more stuff in the syntax that isn't in common use. At the time the standard was written, it was considered important for users to be able to route their own mail, hopping it from machine to machine. The widespread takeup of the Internet where domain addresses for mail are distributed using the DNS has superseded the requirement. Because of the widespread use of spam mail, many sites are no longer prepared to relay mail for random third parties either.

Finally

If you look around your system for programs that are intended to understand the text that you type, then you'll find tokenization and parsing rules. You'll often find that the syntax contains ways to ensure that the program can unambiguously decode what the user has typed. The design of the input is often a trade-off between what is easy for the system to comprehend and what "feels natural" for the human.

I recommend you take a look at RFC 822. RFC's are widely distributed, I got my copy from `ftp://ftp.uu.net//.vol/2/inet/rfc/rfc822.Z`, which mirrors `ftp://nic.ddn.mil/rfc`.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.