Lexing and SCM

CS565

Purdue University

January 18, 2010

Lexing

```
print("Hello!")
```

→

IDENTIFIER(print)
LEFT_PAREN
STRING_LITERAL(Hello!)
RIGHT_PAREN

CS565 Lexer

`https://codu.org/projects/cs565s10/hg/`

Code to know:

- unicode.h
- lex.h
- jstokens.h

A brief primer on Unicode

- (Note: You do not need to learn Unicode, this is just knowledge for your benefit to understand the lexer!)
- Characters are indexes into a (giant, sparse) array
- That index is a codepoint
- ASCII is codepoints 0-127
- UTF-8 encodes ASCII as ASCII (whew!)
- The rest takes multiple bytes
- Stored in the lexer as 32-bit integers (unicodept *)

Lexer — jsTokenizeu8

```
struct Buffer_JavaScriptToken jsTokenizeu8
(char *u8str,
 char *file,
 unsigned int line,
 unsigned int col);
```

Lexer — JavaScriptToken

```
struct JavaScriptToken {
    int token;
    char *file;
    unsigned int line, col, pts;
    unicodept *uni;
    int32_t i, e;
    double d;
};
```

- Token enumeration is in jstokens.h
- line, col are 0-indexed
- pts is size (number of unicode codepoints)
- $(i + d) * 10^{e} =$ value for NUMERIC_LITERAL tokens

Lexer — LINE_TERMINATOR

- Token stream includes LINE_TERMINATOR
- Usually used as whitespace
- But sometimes used as a semicolon (welcome to JavaScript)

Lexer — jsGetNextToken

```
struct JavaScriptNextToken jsGetNextToken(
        struct Buffer_JavaScriptToken toks,
        size_t from,
        int slineterminators);
struct JavaScriptNextToken {
    /* the token requested */
    struct JavaScriptToken *tok;
    /* the index of the next token */
    size_t next;
    /* was there a line terminator? */
    int lineterminator;
};
```

SCM Tools

- Decentralized
  - **Mercurial**
  - git
  - GNU Arch/Bazaar (bzr), darcs, Perforce, ...
- Centralized
  - Subversion (svn)
  - CVS
  - RCS, SCCS (CSSC), ...

Mercurial

(Live demo)

Build Tools

- **make**
- autoconf (et al)
- cmake, scons, ...

Note: I can be flexible with the build system you use, so long as building is automated and doesn't require a GUI application. Talk to me first.

Makefiles

- The theory: Rules on how to get from one type of file to another
- e.g. .c → .o and .o → binary

Makefile rules

```
hello: hello.o
	cc hello.o -o hello

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

(Note: Indentation *must* be tabs, not spaces)

Dependencies

```
hello: hello.o
        cc hello.o -o hello

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

hello.c: hello.h
```

Variables

```
CC=gcc
CFLAGS=-O2 -g
LD=$(CC)
LDFLAGS=

hello: hello.o
        $(LD) $(CFLAGS) $(LDFLAGS) hello.o -o hello

.c.o:
        $(CC) $(CFLAGS) -c $< -o $@

hello.c: hello.h
```

Variables may be overridden:

```
$ make CC=pcc
```

Special build rules

```
CC=gcc
CFLAGS=-O2 -g
LD=$(CC)
LDFLAGS=

HELLO_OBJS=hello.o

all: hello

hello: $(HELLO_OBJS)
        $(LD) $(CFLAGS) $(LDFLAGS) $(HELLO_OBJS) -o hello

.c.o:
        $(CC) $(CFLAGS) -c $< -o $@

hello.c: hello.h

clean:
```

Example

(Live demo)

Parsing

- Recursive-descent parser
- No parser generators: Write your parser by hand
- Vagaries of parsing JavaScript discussed next class