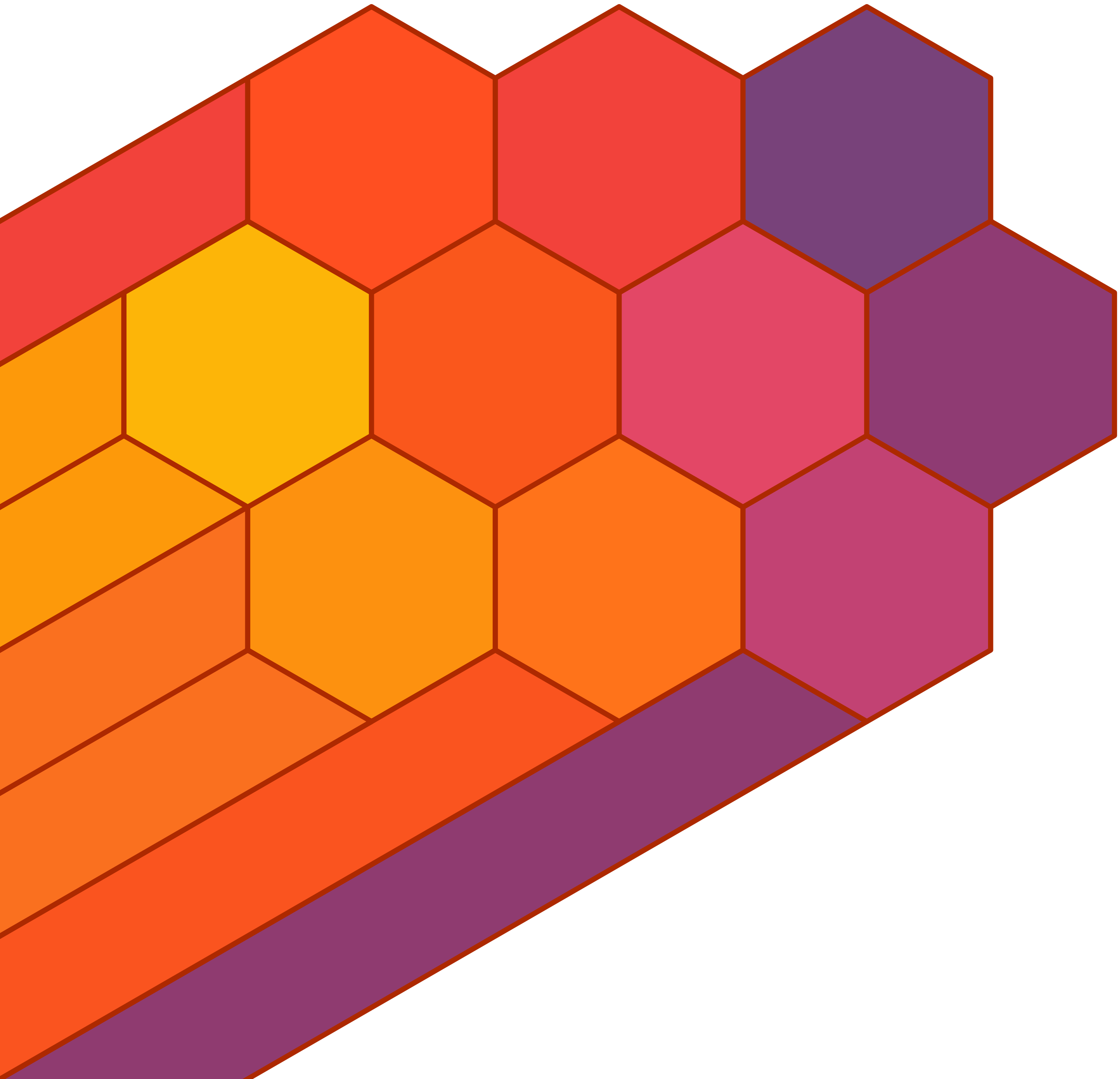
ucsd-psystem-xc
UCSD p-System Cross Compiler

Reference Manual

Peter Miller
<pmiller@opensource.org.au>



.

This document describes ucsd-psystem-xc version 0.13
and was prepared 12 November 2012.

NAME

ucsd-psystem-xc − UCSD p-System Pascal cross compiler

DESCRIPTION

The *ucsd-psystem-xc* package is a collection of tools for compiling Pascal source files to produce UCSD p_System code files. The package includes:

ucsdpsys(1)
: A laucher to run the virtual machine comfortably from the command line. It includes a batch mode for automating (scripting) operations.

ucsdpsys_assemble(1)
: The cross assembler. It is able to assemble several different target microprocessor architectures in the one executable.

ucsdpsys_compile(1)
: The cross compiler. It understands the UCSD Pascal dialect, including UNIT definitions and references.

ucsdpsys_depends(1)
: May be used to determine include file dependencies, for use with *make*(1) and other build tools.

ucsdpsys_disassemble(1)
: For disassembling UCSD p-System code files. This is used to verify the correctness of the compiler.

ucsdpsys_downcase(1)
: A untility for converting Pascal code to lower case, leaving string constants and comments unaltered.

ucsdpsys_errors(1)
: A utility to translate back and forth between text and binary representations of the assembler error message files.

ucsdpsys_libmap(1)
: A utility for printing segment maps of UCSD p-System library files.

ucsdpsys_librarian(1)
: A utility for manipulating the segments within UCSD p-System codefiles.

ucsdpsys_link(1)
: A utility for linking UCSD p-System codefiles to their assembler components.

ucsdpsys_opcodes(1)
: A utility to translate back and forth between text and binary representations of the assembler opcode files.

ucsdpsys_setup(1)
: A utility to translate back and forth between text and binary representations of the `system.miscinfo` file.

Sister Projects

Some other projects will be of interest to you.

ucsd-psystem-fs
: This package contains tools for manipulating UCSD p-System floppy disk images, and a file system for mounting them in Linux as real file systems.
http://ucsd-psystem-fs.sourceforge.net/

ucsd-psystem-os
: This project provides a self-hosting set of system sources. You need the disk images produced by this project for the virtual machine to have a “system.pascal” file to run (this provides runtime support and the user command executive). This is a work in progress.

uvsd-psystem-vm
: This package provides a fully featured UCSD p-Machine emulator.

ARCHIVE SITE

The latest version of *ucsd-psystem-xc* is available on the Web from:

URL:	http://ucsd-psystem-xc.sourceforge.net/	
File:	ucsd-psystem-xc-0.13.README	# Description, from the tar file
File:	ucsd-psystem-xc-0.13.lsm	# Description, LSM format
File:	ucsd-psystem-xc-0.13.tar.gz	# the complete source
File:	ucsd-psystem-xc-0.13.pdf	# Reference Manual

BUILDING ucsd-psystem-xc

Full instructions for building *ucsd-psystem-xc* may be found in the *BUILDING* file included in this distribution.

COPYRIGHT

AUTHOR

Peter Miller	E-Mail:	pmiller@opensource.org.au
/\/*	WWW:	http://miller.emu.id.au/pmiller/

RELEASE NOTES

This section details the features and bug fixes of each of the releases.

Version 0.13 (2012-Nov-12)

- The *ucsdpsys_charset*(1) command now understands more font file formats, including Terak and PSF Tools format.
- The *ucsdpsys_charset*(1) command has a new **−negative** option, that is used to calculate the inverted “top half” of a font from the normal “bottom half”, as is common in Terak fonts.
- The *ucsdpsys_charset*(1) is now able manipulate the boot logo in Terak system.charset files.
- The *ucsdpsys_osmakgen*(1) command now takes advantage of recent *ucsdpsys_osmakgen*(1) features.
- There is a new *ucsdpsys_foto*(1) command, used to convert UCSD Pascal `.foto` files into `.png` files, and *vice versa*.

Version 0.12 (2012-Nov-02)

- The *ucsdpsys_osmakgen*(1) command now understands how to process SYSTEM.CHARSET files.
- There is a new *ucsdpsys_osmakgen*(1) option called **`--arch-from-host`**, that may be used to translate a host name (*e.g.* “Terak”) into an arch name (*e.g.* “pdp11”).
- The *ucsdpsys_charset*(1) command now understands a **`--architecture=terak`** option, which means to work on a SYSTEM.CHARSET file suitable for a Terak system, where the glyphs are 8x10 and laid out differently in the binary file.

Version 0.11 (2012-Jul-28)

- Kai Henningsen <kai.extern@gmail.com> discovered that ’Makefile’ files generated by ucsdpsys_osmakgen did not correctly support the ’distclean’ target. This has been corrected.
- Work is in progress to be able to cope with multiple p-machine versions.
- The compiler is now able to cope with variables declared in plain units.
- The *ucsdpsys_osmakgen*(1) command now understands how to generate the necessary debian/ files for building a debian package from the ucsd p-system operating system sources.
- Thw *ucsdpsys*(1) file no longer creates the implied system disk image if one of the supplied disk images is a functioning system disk.
- The *ucsdpsys*(1) command now better understands where ucsd-psystem-os installs its files, which it needs in order to build the default system disk image.

Version 0.10 (2011-May-18)

- A bug which caused a segfault in the *ubsdpsys −−batch* option has been fixed.
- The *ucsdpsys_osmakgen*(1) command, used by the ucsd-psystem-os project to generate its `Makefile`, now understands the presence of man pages, and installs them appropriately.

Version 0.9 (2011-Feb-02)

- The slides of the LCA 2011 talk "Factory Factory Factories" is now available in the web site.
- The *ucsdpsys_osmakgen*(1) command has been improved, with a view to Debian packaging of the OS.
- The *ucsdpsys*(1) command has a new **--no-system** option, to suppress the construction of a system disk image.
- There is a new *ucsdpsys_compile*(1) option, **--library-path** for adding directories to the library search path.
- The *ucsdpsys_compile*(1) command now fully supports the `(*$U` *filename* `*)` control comment.
- The *ucsdpsys_assemble*(1) command now understands the `.error .print .sbttl .title` pseudo-ops, mostly named for PDP-11 assmebler pseudo-ops of the same name.
- The *ucsdpsys_charset*(1) command has been moved to this project, out of the ucsd-psystem-fs project.
- The *ucsdpsys_assemble*(1) command now understands how to produce assembler listings, using the **-L** option. See *ucsdpsys_assemble*(1) for more information.
- The *ucsdpsys_compile*(1) command now issues warnings for unreachange statements. There is a new `(*$warning unreachable false *)` control comment to disable the warning.
- The project download web page now includes a link to the LunchPad PPA, where pre-compiled Ubuntu packages are available.
- The *ucsdpsys_assemble*(1) command now understands the `.ref` pseudo-op, and generates the appropriate relocation information.
- The *ucsdpsys_assemble*(1) command now more closely emulates the UCSD native assembler, in the way it forgets symbols created between one `.proc` and another. This stops historical source files from complaining about multiply defined symbols all over the place.
- The *ucsdpsys_assemble*(1) command now requires that the architecture be explicitly stated, either with the `.arch` pseudo-op, or the **--arch** command line option, in all cases.
- The *ucsdpsys_assemble*(1) command now ignores all input after the `.end` directive.
- The *ucsdpsys_assembler*(1) command now understands `.gt` greater than, >= greater than or equal, `.lt` less than, <= less than or equal, <> inequality, and = equality comparisons.
- The *ucsdpsys_assemble*(1) command, now understands, for 6502 opcodes, how to relocate segment relative addresses for absolute addressing opcodes.
- The *ucsdpsys_assemble*(1) command now understands conditional assembly `.if`, `.else` and `.endc` pseudo-op directives.
- The *ucsdpsys_assembler*(1) command now understands the `.macro` pseudo-op, for defining an substituting macros into the code stream.
- A bug has been fixed in the code that checks codefiles for validity. It no longer rejects segment dictionaries with zero-length UNITSEG segments. These are produced when a program USES a non-intrinsic unit, but is not yet linked.

Version 0.8 (2010-Aug-28)

- The *ucsdpsys_assemble*(1) cross assembler now understands the `.func` pseudo-op.
- The error message formatting has been changed to use a 4 character hanging indent for multi-line error messages.
- A bug has been fixed in the *ucsdpsys_osmakgen*(1) command, it now correctly understands how to remove system segments from libraries with an assembler component.
- The *ucsdpsys_osmakgen*(1) command now understands how to link Pascal programs with their assembler components.
- A bug has been fixed in the *ucsdpsys*(1) command, it no longer fails if its temporary files are unlinked twice.
- There is a new *ucsdpsys_compile*(1) **--view-path** option, symmetric with the *ucsdpsys_assemble*(1) and *ucsdpsys_depends*(1) commands' options of the same name.
- The *ucsdpsys_assemble*(1) command now understands the `.incude` pseudo-op. This is also a new corresponding **-I** command line option.
- A bug has been fixed in the *ucsdpsys_librarian*(1) command, it now patches the segment number in the procedure dictionary when it renumbers a segment.
- A bug has been fixed in the *ucsdpsys_disassemble*(1) and *ucsdpsys_libmap*(1) commands, they were printing SEPPROC link information incorrectly.
- The *ucsdpsys_osmakgen*(1) command now generates an "install" target, so that the results of the build can be installed into the system.
- The *ucsdpsys_assemble*(1) cross assembler now groks unary minus (−e) unary plus (+e) bit-wise and (e1 & e2), bit-wise or (e1 | e2), bit-wise not (˜e), bit-wise exclusive-or (e1 ˆ e2), and modulo (e1 % e2) expressions.
- The *ucsdpsys_compile*(1) cross compiler can now cope with VAR clauses in the IMPLEMENTATION section of a UNIT.
- The *ucsdpsys_compile*(1) cross compiler is now able to cope with units that export variables, noth intrinsic and non-intrinsic.
- The *ucsdpsys_compile*(1) grammar now understands "var anything" parameters to external assembler procedures and functions.
- The *ucsdpsys_osmakgen*(1) command now understands assembler source file include dependencies.
- The *ucsdpsys_depends*(1) command now understands how to process assembler source files, when looking for include dependencies.
- The *ucsdpsys_assemble*(1) command now procuces minimally correct relocation data sectiosn for each native code procedure. The *ucsdpsys_disassemble*(1) command now has a minimally correct understanding of relocation data.
- There is a new *ucsdpsys_link*(1) command, that may be used to link programs and libraries of separate procedures and functions together, to produce executable output codefiles. See *ucsdpsys_link*(1) for more information.
- The *ucsdpsys_libmap*(1) and *ucsdpsys_disassemble*(1) commands now include the EOFMARK link information record, to be sure it contains the correct argument. The *ucsdpsys_assemble*(1) and *ucsdpsys_compile*(1) commands now correctly generate EOFMARK link information records.
- The *ucsdpsys_littoral*(1) command now correctly translates `nil` to `NULL`.
- The *ucsdpsys_littoral*(1) command now expands `with` variables completely. This preserves the semantics into the C++ code.
- There is now a build dependency on the *libexplain* project (http://libexplain.sourceforge.net/).
- A bug has been fixed in the *ucsdpsys*(1) command, it no longer overwrite its own temporary files. All of the *ucsdpsys*(1) options now have long versions as well. The UCSD p-System volumes that are created

on-the-fly are now created large enough to hold all of the data.

- The *ucsdpsys_osmakgen*(1) command is now able to figure out when it needs to make a copy of `system/globals.text` based on include dependency information and the source file manifest.
- The `for` statement now understands `real` control variables. Note that the native compiler does not allow this.
- The *ucsdpsys_assemble*(1) cross assembler now understands the `.def` pseudo-op.

Version 0.7 (2010-Jun-21)

- There is a new *ucsdpsys_osmakgen*(1) command, used to write the `Makefile` for the ucsd-psystem-os project.
- The *ucsdpsys_setup*(1) command now accepts an `--`**arch** option, in order to select the byte sex of the `SYSTEM.MISCINFO` file it generates.
- There is a new *ucsdpsys_errors*(1) command, to translating the assembler error files from text to binary.
- The *ucsdpsys_opcode*(1) command now understands the opcode file format used by the UCSD Adaptive Assembler.
- A bug has been fixed in the *ucsdpsys_depends*(1) command, it no longer writes to a file called "`–`" when it should write to the standard output.
- The *ucsdpsys_librarian*(1) command has a new `--remove-system-segments` option, used to remove dummy segments from a `(*$U-*)` utility.
- The *ucsdpsys_librarian*(1) command is now able to renumber segments when they are transferred bwtween codefiles.
- The *ucsdpsys_compile*(1) command has a new `--`**host** option, that allows you to set the byte-sex based on the name of the host. Which helps those of us who don't necessarily remember what endian-ness all of the hosts actually are.
- The *ucsdpsys_assemble*(1) command has a new `--`**architecture** option, to permit the target architecture to be set from the command line.
- The *ucsdpsys_assemble*(1) multi-target cross assembler now has the beginnings of support for PDP-11 assembler.
- The cross compiler is now able to recognize the ord/odd hack (used to gain access to bit-wise opcodes) and turn such expression trees from logical operations into bit-wise operations.
- The disassembler no longer rejects valid machine code segments with very short procedures.
- The *ucsdpsys_assemble*(1) multi-target cross assembler now has beginnings of 6502 support, including both the MosTech syntax and the Apple syntax.
- A bug has been fixed in the cross compiler, it now generates the correct opcode for the inline-math sqrt function.
- The assembler now has a `.radix` pseudo-op, that may be used to change the default radix being used by the assembler.
- A bug has been fixed in repeat/until statements, it was generating no code in some cases.

Version 0.6 (2010-May-30)

- The compiler now understands EXTERNAL function and procedure declarations, and produces corresponding linker records.
- The compiler now has complete long integer support.
- The compiler now understands the built-in STR function.
- It is now possible to write long integer constants in the source code. They take the same forms as other integer constants, except they are suffixed with the letter L. This is an idea transplanted from C, the UCSD native compiler does not recognise such constants. It makes testing and debugging the long integer constant folding much easier.
- The compiler now understands `unit` definitions, using II.1 syntax and semantics. If II.0 `separate unit` definitions are seen, they result in a warning, and the `separate` keyword is otherwise ignored.
- The compiler now understands a C-style ternary operator expression `(e1 ? e2 : e3)`. The UCSD native compiler doesn’t have this.

Version 0.5 (2010-May-17)

- There is a new `(*$feature underscore-significant true*)` contol comment, that may be used for increased ISO 10206 conformance.
- A bug has been fixed in the RECORD code, it no longer places the selector variable in the variant part of the record, and thus is no longer requesting memory from NEW that is one word short.
- There is a new `(*$feature efj-nfj false*)` control comment to turn off the use of the EFJ and NFJ opcodes.
- There is a new `(*$feature short-with false*)` control comment, that can be used to turn off WITH statement optimizations.
- The built-in UNITWRITE procedure now accepts string constants for the second parameter. The UCSD native compiler did not allow this. Handy for debugging the system I/O procedures.
- The compiler now optimizes IF stratements with GOTO clauses. It now goes directly to the label from the condition, when possible, rather than using UJP in the individual clauses.
- The IF statement now generates better code for the case where THEN is empty but ELSE is not.
- The compiler now understands the ISO 10206 integer constants with an explicit radix. This was not available in the UCSD native compiler, for obvious reasons.
- The is a new *ucsdpsys_setup*(1) command, used to encode and decode the `SYSTEM.MISCINFO` file.
- There is a new *ucsdpsys_downcase*(1) command, that may be used to convert identifiers in Pascal source code from upper case to lower case.
- The compiler no longer has a problem with sets passed as parameters. The way sets are push onto the stack has been further optimized.
- The compiler now understands how to optimize away MOVELEFT, MOVERIGHT and FILLCHAR with a constant zero or negative length.
- A bug has been fixed in the IN operator, in the case where the set had a fixed size.
- A bug has been fixed in the constant folding of string comparisons, it was getting relational comparisons (<, <=, >, >=) wrong, but equality comparisons (=, <>) right.
- A bug has been fixed in the indexing of byte arrays (pointers) with enum types. It no longer throws an assert.
- The compiler now issues warnings for comments that are not ISO 7185 comforming.
- A bug has been fixed in the code generation of MOV opcodes, in the case where more than 127 words had to be moved.
- The compiler now understands `arctan` (ISO 10206) as a synonym for `atan`, but only if `(*$feature inline-math true*)` is in effect.

- The compiler now generates correct code for NOT logical expressions assigned to a boolean variable, or passed as a boolean parameter.
- A bug has been fixed in the code that folds constant MPI (integer multiply) expressions.
- A bug has been fixed in the optimization of integer subtraction.
- A bug has been fixed in the optimization of the ADI (add integer) expression.
- A bug has been fixed in the optimisation of the logical NOT expression.
- The cross compiler now understands the bit-wise integer AND, OR and NOT expressions.
- The compiler now generates LDB (load byte) and STB (store byte) instructions for packed arrays of 8-bit things, not just packed array of char. This is the same behaviour as the UCSD native compiler.
- There is a new *ucsdpsys_librarian*(1) **−R** option, that can be used to remove segments by name or by number.

Version 0.4 (2010-May-06)

- A bug has been fixed in the code generation for large set constants.
- The CASE statement now understands negative case values.
- The compiler now understands how to cast string constants into packed-array-of-char constants, when they are procedure and functions parameters.
- The compiler now understands when a case control expression is a function call with no parameters.
- The compiler now understands functions calls with no parameters on either side of the IN operator.
- The compiler now generates the correct code for segment procedures that are declared forward.
- The compiler now understands how to pass parameters that are records, by value.
- The compiler now generates correct code for array parameters when they are passed by value.
- A bug has been fixed in the READLN code generation, it no longer throws an assert.
- The compiler no longer issues syntax errors when semicolons appear in questionable places in RECORD declarations.
- The way symbol conflicts and shadows are calaculated has been changed, it was getting false positive on the conflict tests.
- The compiler now understands passing a string as the first parameter to the FILLCHAR procedure.
- The compiler now understands the unary plus operator.
- The compiler now understands the built-in GET, GOTOXY, PAGE, PUT, SEEK, UNITSTATUS and UNITWAIT procedures.
- There is a new (*$feature inline-math true*) control comment. When this is enabled, the compiler now understands the built-in ATAN, COS, EXP, LN, LOG, SIN and SQRT functions.
- There is a new *ucsdpsys_assemble*(1) command, that may be used to assemble machine code and p-code. It isn’t particularly capable, as yet, but it will become more so as work proceeds on the p-machine validation
- The compiler now accepts for loops of char values where one or both limits are char constants.
- The built-in FILLCHAR procedure now accepts its third paramater being an enumerated type. This is for backwards compatibility with the UCSD native compiler.
- The compiler now understands how to index an array by a char value. Previously it was throwing an assert.
- There is a new (*$feature ignore-undefined-segment-zero true *) option, that can be used to turn off checking for undefined forward declarations, when those symbols would be in segment zero. This “feature” is used by system utilities. All other cases of forward functions being undefined result in a fatal error; use EXTERNAL for procedures to be linked later.

- The disassembler can now cope with broken pointers in a segment's procedure dictionary. Usually undefined (external) procedures with have a zero (0) entry in the procedure table.
- The string parameters length check is now a warning, rather than an error. This is because the implicit copy at run-time will throw a run-time error of the string doesn't fit.
- The compiler now accepts calls to the built-in EOF and EOLN functions with no parameters.
- The code generation for empty set constant has been improved. It no longer throws an assert. The same assert revealed that empty sets as a function parameter was not correctly being cast to the appropriate type of set.

Version 0.3 (2010-Apr-25)

- A warning is now issued if a case statement contains an `otherwise` clause. You can disable the warning by using the `(*$warning otherwise false*)` control comment.
- The compile listing now includes the symbol table for each procedure and function.
- A bug has been fixed in the code that derefereces pointers to strings. It no longer tries to laod the whole string onto the stack. The compiler now understands how to deal with string-typed fields on the right hand side of dot (expr.name) expressions.
- A bug has been fixed where function parameters that were the names of functions that had no parameters were not being called.

The compiler no longer issues duplicate label warnings. In some
cases it was issuing warnings about unused labels twice.

- The compiler now understands the built-in COPY, DELETE, EOF, EOLN, FILLCHAR, INSERT, POS, UNITBUSY and UNITCLEAR functions and procedures.
- The compiler no loger throws an assert if a procedure in segment zero is EXIT()ed.
- The compiler now correctly scopes enumerated constant definitions that are declared within the record scopes.
- A bug has been fixed in the code that copied non-var string parameters into their local temporaries.
- Thw compiler now understands how to perform a non-local function return assignment.
- The compiler now also accepts an integer value as the third parameter of fillchar, even thouh it is documented to take a char value.
- A bug has been fixed where constant negative array indexes would cause an assert to fail. It turned out that some optimizations were not checking the range of offsets, and creating invalid offsets.
- The compiler now understands declaring and accessing arrays using multi dimension syntax.
- A number of error messages concerning forward declared types have been improved; they are now earlier, and less cryptic.
- A bug has been fixed in the code generation of constant sets. They are no longer all-bits-zero, but instead contain the correct value.
- The compiler now only range checks the CHR parameter if requested. The UCSD native compiler did not range check CHR.
- The compiler now checks parameter string lengths (declared *vs* actual) for overruns.
- The compiler now understands about file^ variables.
- The *ucsdpsys*(1) command is now better at cleaning up its temporary files.
- The boolean comparison operators (=, <>, <=, <, >=, >) now have additional code to cope with one side or the other being a constant.
- A bug has been fixed in the way constant folding was handled around the FOR statement's limits.

Version 0.2 (2010-Apr-19)

- The target for this release was to be able to compile the UCSD native Pascal compiler from source. This has been achieved. It has yet to be determined if the compiler thus created actually functions.
- For differences between this cross compiler and the UCSD native compiler, see the *ucsdpsys_compile*(1) man page. The most notable difference is that SIZEOF is a keyword, requiring the UCSD native compiler's PROCEDURE SIZEOF to be renamed.
- Numerous bugs have been fixed, usually in unexplored corner cases.
- The compiler now understands the ABS, BLOCKREAD, BLOCKWRITE, CLOSE, CONCAT, EXIT, HALT, IDSEARCH, IORESULT, KEYBOARD, LENGTH, MARK, MOD, MOVELEFT, MOVERIGHT, OPENNEW, OPENOLD, PWROFTEN, READ, READLN, RELEASE, RESET, REWRITE, ROUND, SCAN, TREESEARCH, TRUNC, UNITREAD, UNITWRITE and WRITELN built-in symbols.
- The STRING type has been turned into a built-in named type. This permits the unwise user to redefine STRING to be a variable or a procedure or a function, or (for maximum confusion) a different type. This is what shadow warnings are for.
- The compiler now understands the CASE, FOR, REPEAT UNTIL and WITH statements.
- The compiler now understands comparisons of CHAR values.
- The compiler now accepts pointers as parameters to the ORD function. This seems oddly inconsistent, in a language as intent as Pascal is, with the protection of the programmer from his own folly.
- The compiler now understands set arithmetic and set comparisons.
- It is now possible, using the *ucsdpsys_compile −−listing* option, to obtain a compiler listing. The listing contains the source code interleaved with the disassembled p-code. The `(*$L)` control comment is ignored.
- The compiler now understands = and <> comparisons of multi-word values (arrays and records).
- The compiler can now be configured to have longer identifier (name) lengths. It defaults to 8 for compatibility, and it still drops underscores.
- The compiler now understands comparisons of packed arrays of char.

Version 0.1 (2010-Apr-01)

First public release.

- The following built-in functions are understood: CHR, MEMAVAIL, ODD, ORD, PRED, SIZEOF, SQR, SUCC, TIME.
- All of the usual Pascal expresion operators are understood, although not always across the full range of parameter types.
- The cross compiler can produce both little-endian codefiles and big-endian codefiles.
- A number of features from modern Pascal implementations are avilable: hex constants, binary constants, short-circuit boolean evaluation, the address-of (@) operator,
- Most of the Pascal statement types are available, including: BEGIN END, CASE (and OTHERWISE), FOR, GOTO (local), IF THEN (ELSE), NEW (including variant types), REPEAT UNTIL, WHILE, WITH, WRITE, WRITELN. It is not yet possble to use non-local GOTO.
- Segment procedures can be created, and UNIT interfaces can be accessed from library codefiles. It is not yet possible to compile UNITs. While FORWARD procedures and functions are understood, EXTERNAL procedures and functions are not yet supported.
- All of the UCSD Pascal data types are supported: ARRAY (including PACKED ARRAY), BOOLEAN, CHAR, enumerated, FILE, INTEGER INTERACTIVE, pointers, REAL, RECORD (including PACKED RECORD), SET, STRING (including STRING[n]), subrange, TEXT. The long integer types are not yet supported.

- The cross compiler understands many of the UCSD Pascal constants, including: FALSE, MAXINT, NIL, TRUE,
- The cross compiler is able to optimize most statements and expressions better than the Apple Pascal native compiler. Constant expressions are folded at compile time.
- There is a *ucsdpsys_depends*(1) command, that can be used by your build system to scan for `(*$I filename*)` include directives.

Version 0.0 (2006-May-22)

No public release.

NAME

How to build ucsd-psystem-xc

BEFORE YOU START

There are a few pieces of software you may want to fetch and install before you proceed with your installation of ucsd-psystem-xc.

Boost Library
: You will need the C++ Boost Library. If you are using a package based system, you will need the *libboost-devel* package, or one named something very similar.
http://boost.org/

libexplain
: The *ucsd-psystem-xc* package depends on the libexplain package: a library of system-call-specific strerror replacements.
http://libexplain.sourceforge.net/

GNU Groff
: The documentation for the *ucsd-psystem-xc* package was prepared using the GNU Groff package (version 1.14 or later). This distribution includes full documentation, which may be processed into PostScript or DVI files at install time − if GNU Groff has been installed.

SITE CONFIGURATION

The **ucsd-psystem-xc** package is configured using the *configure* program included in this distribution.

The *configure* shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates the *Makefile* and *lib/config.h* files. It also creates a shell script *config.status* that you can run in the future to recreate the current configuration.

Normally, you just *cd* to the directory containing *ucsd-psystem-xc*'s source code and then type

```
% ./configure
...lots of output...
%
```

Running *configure* takes a minute or two. While it is running, it prints some messages that tell what it is doing. If you don't want to see the messages, run *configure* using the quiet option; for example,

```
% ./configure −−quiet
%
```

To compile the **ucsd-psystem-xc** package in a different directory from the one containing the source code, you must use a version of *make* that supports the *VPATH* variable, such as *GNU make*. Change directory to the directory where you want the object files and executables to go and run the *configure* script. The *configure* script automatically checks for the source code in the directory that *configure* is in and in .. (the parent directory). If for some reason *configure* is not in the source code directory that you are configuring, then it will report that it can't find the source code. In that case, run *configure* with the option `−−srcdir=`*DIR*, where *DIR* is the directory that contains the source code.

By default, *configure* will arrange for the *make install* command to install the **ucsd-psystem-xc** package's files in */usr/local/bin*, and */usr/local/man*. There are options which allow you to control the placement of these files.

`−−prefix=`*PATH*
: This specifies the path prefix to be used in the installation. Defaults to */usr/local* unless otherwise specified.

`−−exec−prefix=`*PATH*
: You can specify separate installation prefixes for architecture-specific files files. Defaults to *${prefix}* unless otherwise specified.

`−−bindir=`*PATH*
: This directory contains executable programs. On a network, this directory may be shared between machines with identical hardware and operating systems; it may be mounted read-only.

Defaults to *${exec_prefix}/bin* unless otherwise specified.

`--mandir=`*PATH*
: This directory contains the on-line manual entries. On a network, this directory may be shared between all machines; it may be mounted read-only. Defaults to *${prefix}/man* unless otherwise specified.

The *configure* script ignores most other arguments that you give it; use the `--help` option for a complete list.

On systems that require unusual options for compilation or linking that the *ucsd-psystem-xc* package's *configure* script does not know about, you can give *configure* initial values for variables by setting them in the environment. In Bourne-compatible shells, you can do that on the command line like this:

```
$ CXX='g++ -traditional' LIBS=-lposix ./configure
...lots of output...
$
```

Here are the *make* variables that you might want to override with environment variables when running *configure*.

Variable: CXX
: C++ compiler program. The default is *c++*.

Variable: CPPFLAGS
: Preprocessor flags, commonly defines and include search paths. Defaults to empty. It is common to use `CPPFLAGS=-I/usr/local/include` to access other installed packages.

Variable: INSTALL
: Program to use to install files. The default is *install* if you have it, *cp* otherwise.

Variable: LIBS
: Libraries to link with, in the form `-l`*foo* `-l`*bar*. The *configure* script will append to this, rather than replace it. It is common to use `LIBS=-L/usr/local/lib` to access other installed packages.

If you need to do unusual things to compile the package, the author encourages you to figure out how *configure* could check whether to do them, and mail diffs or instructions to the author so that they can be included in the next release.

BUILDING UCSD-PSYSTEM-XC

All you should need to do is use the

```
% make
...lots of output...
%
```

command and wait. When this finishes you should see a directory called *bin* containing several programs.

If you have GNU Groff installed, the build will also create a *etc/reference.ps* file. This contains the README file, this BUILDING file, and all of the man pages.

You can remove the program binaries and object files from the source directory by using the

```
% make clean
...lots of output...
%
```

command. To remove all of the above files, and also remove the *Makefile* and *lib/config.h* and *config.status* files, use the

```
% make distclean
...lots of output...
%
```

command.

The file *etc/configure.in* is used to create *configure* by a GNU program called *autoconf*. You only need to know this if you want to regenerate *configure* using a newer version of *autoconf*.

TESTING UCSD-PSYSTEM-XC

The *ucsd-psystem-xc* package comes with a test suite. To run this test suite, use the command

```
% make sure
...lots of output...
Passed All Tests
%
```

The tests take a few seconds each, with a few very fast, and a couple very slow, but it varies greatly depending on your CPU.

If all went well, the message

```
Passed All Tests
```

should appear at the end of the make.

INSTALLING UCSD-PSYSTEM-XC

As explained in the *SITE CONFIGURATION* section, above, the *ucsd-psystem-xc* package is installed under the */usr/local* tree by default. Use the `--prefix=`*PATH* option to *configure* if you want some other path. More specific installation locations are assignable, use the `--help` option to *configure* for details.

All that is required to install the *ucsd-psystem-xc* package is to use the

```
% make install
...lots of output...
%
```

command. Control of the directories used may be found in the first few lines of the *Makefile* file and the other files written by the *configure* script; it is best to reconfigure using the *configure* script, rather than attempting to do this by hand.

GETTING HELP

If you need assistance with the *ucsd-psystem-xc* package, please do not hesitate to contact the author at

```
Peter Miller <pmiller@opensource.org.au>
```

Any and all feedback is welcome.

When reporting problems, please include the version number given by the

```
% ucsdpsys_compile -V
ucsdpsys_compile version 0.13.D001
...warranty disclaimer...
%
```

command. Please do not send this example; run the program for the exact version number.

COPYRIGHT

ucsd-psystem-xc version 0.13
Copyright © 2006, 2007, 2010, 2011, 2012 Peter Miller

The *ucsd-psystem-xc* package is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

It should be in the *LICENSE* file included with this distribution.

AUTHOR

Peter Miller	E-Mail:	pmiller@opensource.org.au
/\/*	WWW:	http://miller.emu.id.au/pmiller/

NAME

Factory factory factories − Abandon all flow of control Ye who enter here

ABSTRACT

In many cases, allegedly OO code is still highly procedural and imperative, with little advantage taken of the possibilities presented by inheritance and virtual methods. This talk is about delegating flow of control to an unknown future, manufacturing objects that in turn manufacture more objects, of various class relationships. Why is this useful? How do you follow the program logic, especially if the classes haven't even been written yet? How come the combinatorial explosion doesn't make it untestable? Come along and take a trip down the factory**n rabbit hole, a warren several layers deep, inside a compiler.

INTRODUCTION

There is a particular technique used in the ucsd-psystem-xc project to construct and manipulate Abstract Syntax Tree (AST) representations of the Pascal program. Rather than having the tree operations be implemented by procedural code external to the tree, the manipulations are performed by the tree nodes themselves.

A design goal was to be able to re-use the grammar for the Pascal language, so that other static analysis tools could also be written, but having the grammar and symbol table handling remain in common library code. This complicates things, if we are going to have the tree nodes performing all the work, because this would seem to imply that every tree node would include the methods necessary to perform all tasks and re-uses of the grammar. Happily, this is not the case.

This paper is an extension of the earlier *Compilers and Factories* paper.

THE VIRTUAL KEYWORD

The key concept here is the `virtual` keyword in C++. A virtual method is one that can have different implementations in different derived classes. Thus, for our AST node to perform a different operation, it must be a different derived class.

Some Revision

Long, long ago, there was no C++. Examples of AST representations dating from then would often have C declarations like this:

```
struct expr_t
{
    int kind;
    union
    {
        int value;
        struct
        {
            struct expr_t *lhs;
            struct expr_t *rhs;
        } p;
    } u;
};
```

Manipulating these trees would involve a function such as this:

```
int
expr_evaluate(const struct expr_t *ep)
{
    switch (ep−>kind)
    {
    case CONSTANT:
        return ep−>u.value;

    case PLUS:
        return expr_evaluate(ep−>u.p.lhs)
            + expr_evaluate(ep−>u.p.rhs);
    case MINUS:
        return expr_evaluate(ep−>u.p.lhs)
            − expr_evaluate(ep−>u.p.rhs);

    etc
    }
}
```

Each time you wanted to add a new kind of expression node, you had to visit each of these functions, and add another switch case. This can become an expensive maintenance problem, and also lead to version control bottlenecks for the development team.

In order to be able to add code in the future, but not have these problems, it is necessary to split the problem into pieces, using pointers to functions:

```
    expr_evaluate(const struct expr_t *ep)
    {
        return (*ep−>evaluate_method)(ep);
    }
```

This means our struct declaration changed as well

```
    struct expr_t
    {
        int (*evaluate_method)(const struct expr_t *ep);
        union
        {
            int value;
            struct
            {
                struct expr_t *lhs;
                struct expr_t *rhs;
            } p;
        } u;
    };
```

Notice, in particular, that the `kind` member is now gone, replaced by one or more function pointers. In practice, this tends to be a pointer to a struct full of function pointers, one for each task, because this simplifies the creating of new AST nodes.

All of which means that our actual evaluation comes in separate pieces:

```
    int
    expr_constant_evaluate(const struct expr_t *ep)
    {
        return ep−>u.value;
    }

    int
    expr_plus_evaluate(const struct expr_t *ep)
    {
        return expr_evaluate(ep−>u.p.lhs)
            + expr_evaluate(ep−>u.p.rhs);
    }
```

The actual implementation would have these in separate compilation units. Now that we have split this up, it would also be possible to do away with the union, and `malloc` AST nodes of the appropriate size.

If anyone has done this manually, you will know that there is a lot of machinery that needs to be kept in sync. Much of this machinery is done for you by C++, and it also adds some rigor to the types of nodes, avoiding the numerous type casts required when doing the same thing manually. The C++ could would look something like this:

```
class expression
{
public:
    virtual int evaluate(void) const = 0;
};
```

and the implementations

```
class expression_plus:
    public expression
{
public:
    int
    evaluate(void)
        const
    {
        return lhs−>evaluate() + rhs−>evaluate();
    }

private:
    expression *lhs;
    expression *rhs;
};
```

The key thing to notice is that we replaced the `kind` member with a "vtable", and switches on `kind` with virtual methods.

Flow Of Control

Once all of the machinery is in place, adding a new kind of expression AST node simply means deriving a new class, and implementing the appropriate methods, such as *evaluate* in the above example. If you are a new developer on the team, and you didn't see the machinery unfold, and implemented the first few classes, just how the code actually *reaches* your virtual method can be a bit of a mystery.

The first thing to remember is what a `virtual` method is. It is a type-based dispatch mechanism. There many only be a single call to that method in the entire program, and yet there could be tens or hundreds of implementations of that method. There is no voodoo here, no magic. If it were done long-hand, as in the first example, confusion rarely arises. Just think of it as the same thing, only distributed differently amongst the source files.

The second thing to remember is that you often *don't care* how the code is called, because that mechanism has already been debugged. When flow of control does get to you, all you care about is getting your bit right.

Testability

Is using a virtual method inherently more difficult to test than the original C implementation? They both have the same code, doing the same jobs, the code is merely distributed amongst the source files differently. So, no, the testing burden is unchanged. Do not mistake the C++ verbosity for "more stuff to test", and remember that C++ is *very* verbose.

Quite possibly, the separation of functionality by class means that you can have greater confidence that you will not unintentionally break something else in the file, because you are not even editing the same files.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/expression.h` file, and its derived classes may be found in the `lib/expression/`*`derived`*`.h` and *`tool`*`/expression/`*`derived`*`.h` files (the directory hierarchy mirrors the class hierarchy). The parser can be found in the `lib/pascal/grammar.y` file.

THE FACTORY CONCEPT

A factory in this sense is a function that returns new instances of a class. Think of a parser that reads text, parses it into expressions, and returns a pointer to the abstract syntax tree representing the parsed expression. This is an example of a factory.

Imagine that our (vastly simplified) yacc grammar looked like this:

```
expr
    : NUMBER
        {
            $$ = constant_expr_factory($1);
        }
    | IDENTIFIER
        {
            $$ = name_expr_factory($1);
        }
    | expr '+' expr
        {
            $$ = plus_expr_factory($1, $3);
        }
    | expr '−' expr
        {
            $$ = minus_expr_factory($1, $3);
        }
    ;
```

For each kind of expression, we have a factory that can build them for us. They are not especially complicated:

```
expr *
constant_expression_factory(int value)
{
    return new expr_constant(value);
}
```

But why wouldn't we just put the same code into the grammar production {rules}? Because we wanted to re-use the grammar.

VIRTUAL FACTORIES

The grammar can be re-used by more than one translation task if we add a context object, and some virtual methods:

```
expr
    : NUMBER
        {
            $$ = ctx−>constant_expr_factory($1);
        }
    | IDENTIFIER
        {
            $$ = ctx−>name_expr_factory($1);
        }
    | expr '+' expr
        {
            $$ = ctx−>plus_expr_factory($1, $3);
        }
    | expr '−' expr
        {
            $$ = ctx−>minus_expr_factory($1, $3);
        }
    ;
```

And the `ctx` variable is a pointer to

```
class translator
```

```
{
public:
    virtual expr *constant_expr_factory(int) = 0;
    virtual expr *name_expr_factory(int) = 0;
    virtual expr *plus_expr_factory(expr *, expr *) = 0;
    virtual expr *minus_expr_factory(expr *, expr *) = 0;
    etc
};
```

By deriving different `translator` classes, we can have one translator that implements a compiler, one that implements a pretty printer, one that calculates cyclomatic complexity statistics, *etc*.

The compiler translator creates expression tree nodes that have an implementation that compiles the expressions. The pretty printer translator creates different expression tree nodes that have an implementation that prints the expressions out. And other static analysis tools each have their own implementations.

Testability

Does this make programs that use this technique harder to test? The amount of code to be written is the same, and does the same jobs. So, no, the testing burden is unchanged.

However, you have the advantage that the parser is common to all of the tools, and so bug fixes to the parser are inherited by all tools. Change once, test everywhere? Not quite: if you had *n* separate yacc files, all with the same bug, you would have to make *n* identical changes, and re-test *n* tools. Testing burden unchanged, *but* the probability of unintentionally diverging grammars becomes zero.

Flow Of Control

The need to understand the flow of control comes when the developer is testing a new derivation of the `translator` class. The grammar, and its connection to the translator context has already been written and tested, all you need to do is test the newly derived class. Your test cases, then, must exercise each of the new factory methods, one test for each of the expression productions, and flow of control will then enter each of the factory methods.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/translator.h` file, and its derived classes may be found in the `tool/translator/derived.h` files.

FACTORY FACTORIES

The wheels of this context concept would appear to start to come off when we consider assignment expressions. A grammar for a C-like language could look like this:

```
expr
    : IDENTIFIER
        {
            $$ = ctx->name_expr_factory($1);
        }
    | expr '=' expr
        {
            $$ = ctx->assignment_expr_factory($1, $3);
        }
    | expr '+' expr
        {
            $$ = ctx->plus_expr_factory($1, $3);
        }
    ;
```

How does our name expression factory know which side of the assignment it is on? At code generation time, should it emit a load opcode or a store opcode? We don't know... yet. What we do know is that loads are much more likely than stores, so we initially generate expression trees that would perform loads.

But this just pushes the problem into the `assignment_expr_factory` method. In order to figure out what kinds of assignment opcode to use, it would be necessary to figure out what kind of load opcode is present, and generate the corresponding store

```
expression *
translator_compiler::assignment_expr_factory(expression *e1, expression *e2
{
    const expr_load *test1 =
        dynamic_cast<const expr_load *>(e1);
    if (test1)
        return new expr_store(e1−>get_operand(), e2);

    const expr_array_load *test2 =
        dynamic_cast<const expr_array_load *>(e1);
    if (test2)
        return new expr_store_array(e1−>get_lhs(), e1−>get_rhs(), e2);

    yyerror("inappropriate assignment");
    return new expression_error();
}
```

This makes me cringe. Those down-casts have my alarm bells going off. And all those getters so that AST node privates can be groped, ugh! But what alternative is there? To answer that, let’s backtrack for a moment. Our very first example can be re-written like this:

```
int
expr_evaluate(const struct expr_t *ep)
{
    if (ep−>kind == CONSTANT)
        return ep−>u.value;

    if (ep−>kind == PLUS)
        return expr_evaluate(ep−>u.p.lhs)
            + expr_evaluate(ep−>u.p.rhs);

    if (ep−>kind == MINUS)
        return expr_evaluate(ep−>u.p.lhs)
            − expr_evaluate(ep−>u.p.rhs);

    etc
}
```

The chain of `if` statements in `assignment_expr_factory` is a *`switch` in disguise*, a type-based dispatch in disguise. We should be using a virtual method instead.

But in which class should we place the virtual method? Clearly, it isn’t inside the `translator` class, since we tried it there already. The type-based dispatch is based on the expression type, and that is where the virtual method lives, in the `expression` class:

```
expr: expr ’=’ expr
    {
        $$ = $1−>assignment_expr_factory($3);
    }
```

No, no, no, that can’t be right: the `ctx` object doesn’t get any chance to intervene. Except that it does: when it created the left hand side in the first place.

By creating, say, a compiler specific “load” AST node, it also created the assignment factory; they are the same object. There is no way a pretty printer assignment object will ever be created by a compiler load object (unless you deliberately code it that way).

Note, too, that the error-prone down-casts are *gone*, as is the need to grope anyone’s privates. And the code is faster, too, by eliminating the slow down-casts and multiple tests.

The sharp-eyed reader will have noticed that we have omitted the error case. What happens when it goes wrong? The easiest way is to have the common base class aways emit an error complaining about an inappropriate assignment, unless overridden.

```
expression *
expression::assignment_expr_factory(expression *, expression *)
{
    yyerror("inappropriate assignment");
    return new expression_error();
}
```

In summary, our `name_expr_factory` method manufactured an object that, in turn, contains an `assignment_expr_factory` method, used to manufacture more AST nodes. We now have a factory factory.

Testability

My head is starting to explode. Surely *now* there are combinatorial effects on testing!

Well, yes and no. Yes, programming languages by definition are capable of combinatorial effects when it comes to all the ways you can put together different expressions to build different programs; that is unchanged, compilers need *lots* of testing.

And, no, the factory factories do not making the testing burden worse. They are, after all, implementing the same thing, often with the same code, albeit distributed differently amongst the classes.

Flow Of Control

If I'm a developer adding a new type of assignment to an existing complier implemented this way, how to I know when execution will reach my shiny new expression class' `assignment_expr_factory` method? Well, the same way you would have when it was imperative code: write a test with that kind of assigment in it, and hand it to the parser. Remember: you aren't testing the parser part of the code, only your new assignment type (class).

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/expression.h` file, and its tool-specific derived classes may be found in the *`tool`*`/expression/`*`derived`*`.h` files.

FACTORY FACTORY FACTORIES

Now we turn our attention to the `name_expr_factory` method. It's been trying to look all innocent and inconspicuous.

```
expression *
translator_compiler::name_expr_factory(const char *name)
{
    symbol *sp = lookup(name);
    if (!sp)
    {
        yyerror("name unknown");
        return new expr_error();
    }

    const symbol_extern *test1 =
        dynamic_cast<const symbol_extern *>(sp);
    if (test1)
        return new expr_load_extern(sp);

    const symbol_static *test2 =
        dynamic_cast<const symbol_static *>(sp);
    if (test2)
        return new expr_load_static(sp);
```

```
            const symbol_local *test3 =
                dynamic_cast<const symbol_local *>(sp);
            if (test3)
                return new expr_load_local(sp);

            yyerror("can’t use name here");
        }
```

This is another example of a type-based dispatch in disguise. But where does the virtual method belong? Clearly, not in the `translator` class or derivative, we already tried that. Instead, we implement it in the `symbol` class, as follows:

```
        expression *
        translator::name_expr_factory(const char *name)
        {
            symbol *sp = lookup(name);
            if (!sp)
            {
                yyerror("name unknown");
                return new expr_error();
            }
            return sp−>name_expr_factory();
        }
```

We moved the `name_expr_factory` into the `translator` base class, because it is now identical across all derived classes, because it no longer needs to know about compiler-specific classes.

As in the previous section about assignment expressions: doing symbol accesses this way means that the advantages are the same, the testing burden unchanged, and the error handling is the same.

In summary, the `translator::name_expr_factory` method looked up a `symbol` object that, in turn, contains a `name_expr_factory` method, used to manufacture `expression` AST nodes, that in turn contain `assignment_expr_factory` methods, used to manufacture more `expr` AST nodes. We now have a factory factory factory.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/symbol.h` file, and its derived classes may be found in the `lib/symbol/`*`derived`*`.h` and *`tool`*`/symbol/`*`derived`*`.h` files.

FACTORY**4

Have you thought about variable scopes in Pascal? By having different scopes for `program`s and `function`s (because their variables are accessed by different opcodes) when a new variable is declared, you ask the current `scope` to manufacture a new `symbol` instance that... you get the idea.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/scope.h` file, and its derived classes may be found in the `lib/scope/`*`derived`*`.h` and *`tool`*`/scope/`*`derived`*`.h` files.

COPYRIGHT

AUTHOR

Peter Miller	E-Mail:	pmiller@opensource.org.au
/\/*	WWW:	http://miller.emu.id.au/pmiller/