15-411 Compiler Design: Lab 3
Fall 2007

Instructor: Frank Pfenning
TAs: David McWherter and Noam Zeilberger

Test Programs Due: 11:59pm, Thursday, October 18, 2007
Compilers Due: 11:59pm, Tuesday, October 30, 2007
Revision 1: Sun Oct 13, 2007
Revision 2: Tue Oct 16, 2007

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L3*. This language extends *L2* by structs (in the sense of C), pointers, and arrays. That means the main change from the second lab is that you will have to deal with *memory references*. Because pointers on the x86-64 architecture are 64 bits wide, this also means you will have to deal with differently sized data. Again, performance of both the compiler and the compiled code are only minor considerations at this stage. However, the threshold for compilation time will be slightly tighter, so you may need to start to pay attention to some performance issues.

2 Requirements

As for Labs 1 and 2, you are required to hand in test programs as well as a complete working compiler that translates *L3* source programs into correct target programs written in x86-64 assembly language. When encountering an error in the input program (which can be a lexical, grammatical, or static semantics error) the compiler should terminate with a non-zero exit code and print a helpful error message. To test the target programs, we will assemble and link them using `gcc` on the lab machines and run them under fixed but generous time limits.

3 *L3* Syntax

The syntax of *L3* is defined by the context-free grammar in Figure 1. This is an extension of the grammar for *L2* in the sense that a correct *L2* program should still parse correctly if it does not use any of the new keywords. Ambiguities in this grammar are resolved according to the operator precedence table in Figure 2 and the rule that an `else` provides the alternative for the most recent eligible `if`. Comments are as in *L2*.

4 *L3* Runtime Objects

The *L3* language has four kinds of identifiers: those standing for functions (g), those standing for variables (x), those standing for structs (s), and those standing for field names (f). These are in

$$\begin{array}{lcl}
\langle\text{program}\rangle & ::= & \langle\text{struct}\rangle^* \; \langle\text{function}\rangle^* \\
\langle\text{struct}\rangle & ::= & \textbf{struct} \; \langle\text{ident}\rangle \; \textbf{\{} \; [\; \langle\text{ident}\rangle \; \textbf{:} \; \langle\text{type}\rangle \; \textbf{;}]^* \; \textbf{\}} \; \textbf{;} \\
\langle\text{function}\rangle & ::= & \langle\text{type}\rangle \; \langle\text{ident}\rangle \; \textbf{(} \; \langle\text{paramlist}\rangle \; \textbf{)} \; \langle\text{body}\rangle \\
\langle\text{paramlist}\rangle & ::= & \varepsilon \mid \langle\text{ident}\rangle \; \textbf{:} \; \langle\text{type}\rangle \; [\; \textbf{,} \; \langle\text{ident}\rangle \; \textbf{:} \; \langle\text{type}\rangle \;]^* \\
\langle\text{body}\rangle & ::= & \textbf{\{} \; \langle\text{decl}\rangle^* \; \langle\text{stmt}\rangle^* \; \textbf{\}} \\
\langle\text{decl}\rangle & ::= & \textbf{var} \; \langle\text{ident}\rangle \; [\; \textbf{,} \; \langle\text{ident}\rangle \;]^* \; \textbf{:} \; \langle\text{type}\rangle \; \textbf{;} \\
\langle\text{type}\rangle & ::= & \textbf{int} \mid \langle\text{ident}\rangle \mid \langle\text{type}\rangle \; \texttt{*} \mid \langle\text{type}\rangle \; [\,] \\
\langle\text{stmt}\rangle & ::= & \langle\text{simp}\rangle \; \textbf{;} \mid \langle\text{control}\rangle \mid \textbf{;} \\
\langle\text{simp}\rangle & ::= & \langle\text{exp}\rangle \; \langle\text{asop}\rangle \; \langle\text{exp}\rangle \mid \langle\text{exp}\rangle \\
\langle\text{control}\rangle & ::= & \textbf{if} \; \textbf{(} \; \langle\text{exp}\rangle \; \textbf{)} \; \langle\text{block}\rangle \; [\; \textbf{else} \; \langle\text{block}\rangle \;] \mid \\
 & & \textbf{while} \; \textbf{(} \; \langle\text{exp}\rangle \; \textbf{)} \; \langle\text{block}\rangle \mid \textbf{for} \; \textbf{(} \; [\; \langle\text{simp}\rangle \;] \; \textbf{;} \; \langle\text{exp}\rangle \; \textbf{;} \; [\; \langle\text{simp}\rangle \;] \; \textbf{)} \; \langle\text{block}\rangle \mid \\
 & & \textbf{continue} \; \textbf{;} \mid \textbf{break} \; \textbf{;} \mid \textbf{return} \; \langle\text{exp}\rangle \; \textbf{;} \\
\langle\text{block}\rangle & ::= & \langle\text{stmt}\rangle \mid \textbf{\{} \; \langle\text{stmt}\rangle^* \; \textbf{\}} \\
\langle\text{exp}\rangle & ::= & \textbf{(} \; \langle\text{exp}\rangle \; \textbf{)} \mid \langle\text{intconst}\rangle \mid \textbf{NULL} \mid \\
 & & \langle\text{ident}\rangle \mid \texttt{*} \; \langle\text{exp}\rangle \mid \langle\text{exp}\rangle \; \texttt{.} \; \langle\text{ident}\rangle \mid \langle\text{exp}\rangle \; \textbf{[} \; \langle\text{exp}\rangle \; \textbf{]} \mid \langle\text{exp}\rangle \; \texttt{->} \; \langle\text{ident}\rangle \\
 & & \langle\text{unop}\rangle \; \langle\text{exp}\rangle \mid \langle\text{exp}\rangle \; \langle\text{binop}\rangle \; \langle\text{exp}\rangle \mid \\
 & & \langle\text{ident}\rangle \; \textbf{(} \; [\; \langle\text{exp}\rangle \; [\; \textbf{,} \; \langle\text{exp}\rangle \;]^* \;] \; \textbf{)} \mid \\
 & & \textbf{new} \; \textbf{(} \; \langle\text{type}\rangle \; [\; \textbf{[} \; \langle\text{exp}\rangle \; \textbf{]} \;] \; \textbf{)} \\
\langle\text{ident}\rangle & ::= & [\textbf{A-Z_a-z}][\textbf{0-9A-Z_a-z}]^* \\
\langle\text{intconst}\rangle & ::= & [\textbf{0-9}][\textbf{0-9}]^* \qquad\qquad \text{(in the range } 0 \leq \text{intconst} < 2^{32}\text{)} \\
\langle\text{asop}\rangle & ::= & \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \mid \texttt{\&=} \mid \texttt{\^{}=} \mid \texttt{|=} \mid \texttt{<<=} \mid \texttt{>>=} \\
\langle\text{binop}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \\
 & & \texttt{\&\&} \mid \texttt{||} \mid \texttt{\&} \mid \texttt{\^{}} \mid \texttt{|} \mid \texttt{<<} \mid \texttt{>>} \\
\langle\text{unop}\rangle & ::= & \texttt{!} \mid \texttt{\~{}} \mid \texttt{-}
\end{array}$$

The precedence of unary and binary operators is given in Figure 2.
Non-terminals are in ⟨angle brackets⟩, optional constituents in [brackets].
Terminals are in **bold**.

Figure 1: Grammar of *L3*

Operator	Associates	Meaning
`() [] -> .`	left	parens, subscript, field dereference, field select
`! ~ - *`	right	logical not, bitwise not, unary minus, dereference
`* / %`	left	integer times, divide, modulo
`+ -`	left	plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= > >=`	left	comparison
`== !=`	left	equality, disequality (integer or pointer)
`&`	left	bitwise and
`^`	left	bitwise exclusive or
`|`	left	bitwise or
`&&`	left	logical and
`||`	left	logical or
`= += -= *= /= %= &= ^= |= <<= >>=`	right	assignment operators

Figure 2: Precedence of operators, from highest to lowest

separate name spaces, so a function, a variable, and a struct may have the same name without conflict. Reserved words of the grammar (`struct`, `var`, `int`, `if`, `else`, `while`, `for`, `continue`, `break`, `return`, `NULL`, `new`) cannot be used as any kind of name.

The *L3* language has some complexity in its new constructs, dealing with memory (structs, pointers, and arrays). In order to describe these concisely we use an abstract syntax notation. Eliding some expressions which are as in *L2*:

$$\begin{array}{lll} \text{Types} & \tau & ::= \ \mathbf{int} \mid s \mid \tau* \mid \mathbf{void}* \mid \tau[\,] \\ \text{Expressions} & e & ::= \ c \mid \mathbf{null} \mid x \mid *e \mid e.f \mid e[e] \mid e{\rightarrow}e \mid \mathbf{new}\,\tau \mid \mathbf{new}\ \tau[e] \mid \cdots \end{array}$$

The type **void**$*$ is used only for internal purposes as the type of `NULL` and is not directly accessible to the programmer.

We also have to consider the runtime objects created during the execution of a program. We divide these into two classes: small values and large values. Small values can be held in registers, large values must be in memory. We use a for an address of an object in memory and n for an integer.

$$\begin{array}{lll} \text{Small Values} & w & ::= \ a \mid n \\ \text{Large Values} & W & ::= \ \{f_1{=}V_1; \ldots f_n{=}V_n;\} \mid [V_0, \ldots, V_{n-1}] \\ \text{Values} & V & ::= \ w \mid W \end{array}$$

The first kind of large value is a struct with fields $f_1, \ldots, f_n$, the second an array with n elements. On occasion, we need to know the exact size of objects of a given type, $|\tau|$. We compute this as follows from the type:

$$\begin{array}{lcl}
|\mathbf{int}| & = & 4 \\
|\tau*| & = & 8 \\
|\mathbf{void}*| & = & 8 \\
|\tau[\,]| & = & 8 \\
|s| & = & \|f_1{:}\tau_1; \ldots f_n{:}\tau_n;\|
\end{array}$$

The size of a struct, written here as $\|f_1{:}\tau_1; \ldots f_n{:}\tau_n;\|$ is computed by traversing the fields from left to right, adding padding as necessary so that alignment restrictions on the fields are satisfied. **int**'s are aligned at 0 mod 4, small values of type $\tau*$ and $\tau[\,]$ are aligned at 0 mod 8, and structs are aligned according to their most strictly aligned field. Padding may need to be added at the end of a struct so that its total size is a multiple of its most strictly aligned field.

Arrays $[V_0, \ldots, V_{n-1}]$ cannot be directly embedded in a struct or other array, since their size is unknown at compile time. This is the reason that a value of type $\tau[\,]$ is *small*: it is merely the starting address of the array in memory. Arrays must be aligned according to the requirement of their elements. The start of an array is returned by a call to the `malloc` library function which guarantees an address 0 mod 8, which is the strictest alignment required in our language.

Addresses are represented as 8 byte (unsigned) integers in the machine. They are obtained from the run time system which allocates new memory, or by addition (address arithmetic). There is a special address 0 which is never returned by the runtime system and will be used to denote the null pointer. Memory access is represented as $M[a]$, which reads from memory when used as a value and writes to memory when used on the left-hand side of an assignment $M[a] \leftarrow w$. The number of bytes read or written depends on the size of the small value w.

Along similar lines, we write $V[x]$ for the value of a variable x. On the left hand side $V[x] \leftarrow w$ assigns to x; on the right hand side $V[x]$ denotes the current value of x.

The most important judgments we make are typing, written $e : \tau$, and evaluation, written $e \Rightarrow w$. The first is part of the *static semantics*, which must satisfy some additional conditions that are stated informally. The latter constitutes the *dynamic semantics*, but is also only partially formal since evaluating expressions has effects that we only describe informally. As a general convention, expressions must be evaluated from left to right so that any side effects happen in a deterministic order.

We now present both the static and dynamic semantics for each new construct (structs, pointers, arrays) in turn.

Structs

Statically, several properties must be checked for struct declarations.

- Every type identifier s used in a struct declaration or function must be declared with an explicit `struct` s `{...};` somewhere in the file. The order of `struct` declarations is irrelevant, but the syntax restricts the file so that they must precede all function definitions.
- All struct declarations in a file must have distinct names s.
- Field names in a struct declaration must all be distinct. However, different struct declarations may reuse field names.
- Fields of structs can refer to other structs which can in turn refer to other structs and so on. However, any circular reference from a struct to itself following such a chain must go through a pointer or array type.

The main expression construct for structs is $e.f$. It is easily type-checked.

$$\frac{e : s \quad \textbf{struct}\ s\ \{\ldots f{:}\tau; \ldots\}}{e.f : \tau}$$

We write offset(s, f) for the offset of field f is structure s, in bytes. We suggest to compute this information early and store it in a table so that the compiler can access it easily.

The evaluation rule is slightly tricky, and carries a recurring theme. In several circumstances evaluating an expression yields an address of some object in memory. When this object is small, we can return it directly as a small value for further computation. When the object is large, we cannot directly operate on the object, but must return its address instead.

$$\begin{array}{lcll} e.f & \Rightarrow & M[a+k] & \text{if } e \Rightarrow a \text{ and offset}(s,f) = k \\ & & & \text{where } e : s \text{ and } \textbf{struct}\ s\ \{\ldots f{:}\tau; \ldots\} \text{ and } \tau \text{ small} \\ & \Rightarrow & a + k & \text{as above, except that } \tau \text{ large} \end{array}$$

We also have the derived form $e{\to}f$, which can be desugared into $(*e).f$.

Pointers

The typing for pointer operations are straightforward.

$$\frac{e : \tau*}{*e : \tau} \qquad \frac{}{\textbf{new}\ \tau : \tau*} \qquad \frac{}{\textbf{null} : \textbf{void}*}$$

Unfortunately, the typing of the **null** pointer present a challenge, since we would like to type expressions such as `p == NULL` or statements such as `p->f = NULL` where the left-hand side has some type $\tau*$. In such situations we would like to implicitly convert the type of **null** from **void**$*$ to $\tau*$. We can capture the allowable implicit conversions with the judgment $\tau \leq \sigma$ for two types τ and σ. It means that an object of type τ can be converted to (or viewed as) a value of type σ. In our simple situation, it is defined by only two rules.

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\textbf{void}* \leq \tau*}$$

These are used only in a few places:

$$\frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1\ \texttt{==}\ e_2 : \textbf{int}} \qquad \frac{e_1 : t_1* \quad e_2 : t_2* \quad t_1* \leq t_2*}{e_1\ \texttt{==}\ e_2 : \textbf{int}} \qquad \frac{e_1 : t_1* \quad e_2 : t_2* \quad t_2* \leq t_1*}{e_1\ \texttt{==}\ e_2 : \textbf{int}}$$

where t_i in the last two rules is either a type τ or **void**. There are corresponding rules for disequality `!=`, but no rules for other types. This means we cannot compare arrays or structs for equality, only integers and pointers.

The other places a type conversion may occur is in assignment discussed below, and in function calls and returns. Assume a function g has prototype

$$\tau g(x_1{:}\tau_1, \ldots, x_n{:}\tau_n);$$

Then we have

$$\frac{e_i : \tau_i' \quad \tau_i' \leq \tau_i \quad (1 \leq i \leq n)}{g(e_1, \ldots, e_n) : \tau}$$

$$\frac{e : \tau' \quad \tau' \leq \tau \quad (\textbf{return} \text{ in body of } g)}{\textbf{return}\ e\ \textit{valid}}$$

where s *valid* indicates that the statement s is valid.

The operational semantics is much simpler.

$$\begin{array}{lcll} *e & \Rightarrow & M[a] & \text{if } e \Rightarrow a \text{ with } e : \tau* \text{ and } \tau \text{ small and } M[a] \text{ allocated} \\ & \Rightarrow & a & \text{as above, except that } \tau \text{ large} \\ \mathbf{new}\ \tau & \Rightarrow & a & \text{if } M[a], \ldots, M[a + |\tau| - 1] \text{ are freshly allocated locations} \end{array}$$

Note that we cannot dereference an object of type **void**$*$, but **null** may have been converted to another type so it is entirely possible that we try to derefence the null pointer. The meaning of this is "undefined".

Arrays

Arrays are similar to pointers.

$$\frac{e_1 : \tau[\,] \quad e_2 : \mathbf{int}}{e_1[e_2] : \tau} \qquad \frac{e : \mathbf{int}}{\mathbf{new}\ \tau[e] : \tau[\,]}$$

$$\begin{array}{lcll} e_1[e_2] & \Rightarrow & M[a + n|\tau|] & \text{if } e_1 \Rightarrow a \text{ and } e_2 \Rightarrow n \text{ with } e_1 : \tau[\,],\ \tau \text{ small, } M[a + n|\tau|] \text{ allocated} \\ & \Rightarrow & a + n|\tau| & \text{as above, except that } \tau \text{ large} \\ \mathbf{new}\ \tau[e] & \Rightarrow & a & \text{if } e \Rightarrow n \text{ and } M[a], \ldots, M[a + (n-1)|\tau|] \text{ are freshly allocated} \end{array}$$

Assignment

Assignment is now more general, because the left-hand side need not be a variable, but could be a more complicated expression so we can write, for example, `*p = *q` to copy the contents of q to the location denoted by p. The left-hand side of an assignment must be a so-called *lvalue* ("left value"), which is a certain kind of expression. We define lvalues, assuming that the expression is already known to be well-typed.

$$\text{Lvalues} \quad v \quad ::= \quad x \mid *e \mid v.f \mid e_1[e_2]$$

For an assignment to be a valid statment, the lvalue on the left-hand side must have a small type.

$$\frac{v : \tau_1 \quad e : \tau_2 \quad \tau_2 \leq \tau_1 \quad \tau_1\ \mathit{small}}{v = e\ \mathit{valid}}$$

The type of the right-hand side may be a subtype of the left-hand side to allow, for example, `p = NULL` to be valid if p : **int**$*$.

We also have a new statement which is just an expression e which is evaluated for possible effects. e may have any type, and the computed value is discarded.

To show how assignment evaluates we use some notational slight of hand.

$$\begin{array}{ll} v = e & \text{has the effect of } V[x] \leftarrow w \\ & \text{if } e \Rightarrow V[x] \text{ and } e \Rightarrow w \\ & \text{has the effect of } M[a] \leftarrow w \\ & \text{if } v \Rightarrow M[a] \text{ and } e \Rightarrow w \end{array}$$

Because we check that v is an lvalue and that the type of e is small, this should cover all possible cases.

The meaning of complex assignment operators now also changes and is no longer a syntactic expansion because expressions can have side effects. An assignment

$$v \; op\texttt{=} \; e$$

should execute as

$$\begin{array}{ll} V[x] \leftarrow V[x] \; op \; w & \text{if } v \Rightarrow V[x] \text{ and } e \Rightarrow w \\ M[a] \leftarrow M[a] \; op \; w & \text{if } v \Rightarrow M[a] \text{ and } e \Rightarrow w \end{array}$$

Functions

Regarding functions, several properties must be checked.

- Every function that is called must be declared somewhere in the file. The order of the functions in the file is irrelevant, but a function may not be declared more than once.
- Functions must be called with the correct number of arguments of the correct type.
- Functions must terminate with an explicit `return` statement. This is checked by verifying that each control flow path originating at the top of a function ends in a `return` statement. See a more detailed explanation in the description of *L2*.
- There must be a function `_l3_main()` which returns an integer as the result of the overall computation (and not an exit status).
- Each `break` or `continue` statement must occur inside a `for` or `while` loop.

Variables

Regarding variables, we need to check two properties.

- Every variable must be declared, either as a function parameter or an explicit local variable with a `var` declaration. Function parameters and local variables are local to a function and have nothing to do with parameters or local variables in other functions. Variables may not be redeclared, that is, names of parameters to a given function and its local variables must all be pairwise distinct. There are no global variables.
- Any variable, either a function parameter or locally declared variable, must have a small type. Variables of large type are not permitted. This simplifies parameter passing considerably.

Local variables declared with `var` will be silently initialized to 0 by the compiler if they are integers. Pointers and arrays need not be initialized (and there are no struct variables). Also, the memory locations returned by calls to `new` need not be initialized.

Unlike *L2*, the lack of enforced initialization and also the lack of bounds check in this language means that there are programs in *L3* whose result is not defined. Such programs are considered to be "erroneous", although the compiler must permit them. For your test cases, you may not submit programs whose behavior is undefined because there is no way to check their (in)correctness. Of course, the reference compiler cannot verify whether your programs are erroneous in this sense, so it is up to you to ensure that.

Overloaded Operators

Equality (`==`) and disequality (`!=`) are overloaded, applying to both integers and pointers. See the discussion above on pointers. Operationally, the compiler will have to choose a 4 byte or 8 byte comparison for the two versions, which suggests after type-checking and during translation to intermediate code there should be two explicitly different operators.

5 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for *L3* that produces correct target programs written in Intel x86-64 assembly language. When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Test Files

Test files should have extension `.l3` and start with one of the following lines

`#test return` *i*	program must execute correctly and return *i*
`#test exception`	program must compile but raise a runtime exception
`#test error`	program must fail to compile due to a *L3* source error

followed by the program text. All test files should be collected into a directory `test/` (containing no other files) and bundled as a tar file `tests.tar` with

```
% tar -cvf tests.tar tests/
```

to be submitted via the Autolab server.

Your test programs must test the new features of *L3*: structs, pointers, and arrays. Since the language is backward compatible except for conflicts with reserved words, we may also use the *L2* test cases for regression testing. We cannot test programs whose results are undefined. It is therefore critical that your test programs do not perform any operations whose meaning is undefined. This includes dereferencing the null pointer or accessing an array out of bounds.

We would like some fraction of your test programs to compute "interesting" functions; please briefly describe such examples in a comment in the file.

Compiler Files

The files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a description of your code and algorithms used at the beginning of this file. This will be a crucial guide for the grader.

Issuing the shell command

```
% make l3c
```

should generate the appropriate files so that

```
% bin/l3c <args>
```

will run your *L3* compiler. The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

In order to keep the files you hand in to a reasonable size, please clean up the directory and then bundle it as a tar file `compiler.tar`, for example with

```
% cd compiler
% make clean
% cd ..
% tar -cvf compiler.tar --exclude .svn compiler/
```

to be submitted via the Autolab server. If you are using `cvs` instead of subversion, you may want to use the switch `--exclude CVS` instead of `--exclude .svn` for a cleaner hand-in.

What to Turn In

Hand-in on the Autolab server:

- At least 20 test cases, at least two of which generate an error and at least two others raise a runtime exception. Submit `tests.tar` with the directory `tests/` with only the test files via the Autolab server. The server will test your test files and notify you if there is a discrepancy between your answer and the outcome of the reference implementation. If you feel the reference implementation is in error, please notify the instructors. The compiled binary for each test case should run in 2 seconds with the reference compiler on the lab machines; we will use a 5 second limit for testing compilers.

  Test cases are due **11:59pm on Thu Oct 18**.

- The complete compiler. Submit `compiler.tar` with the directory `compiler/` after applying a `make clean`. The Autolab server will build your compiler, run it on all extant test files, link the resulting assembly files against our runtime system (if compilation is successful), execute the binaries (each with a 5 second time limit), and finally compare the actual with the expected results.

  Compilers are due **11:59pm on Tue Oct 30**.

6 Notes and Hints

There is no address-of operator (`&`), and variables (be it local variables or procedure parameters) cannot hold large values. The suggested implementation therefore is to allocate structs and arrays on the heap. Our runtime system provides the function `void* malloc(size_t nbytes);` for this purpose. Your compiled code should call this function as necessary, assuming `size_t` is `unsigned int` (4 bytes).

There is no corresponding `free`, since we use a conservative garbage collector to reclaim storage.

To check that struct declarations are well-founded, we suggest a simple cycle-checking depth-first search which is in any case necessary to compute field offsets. Global tables to keeps struct sizes, field offsets, and the types of functions seem like an appropriate strategy.

Data now can have different sizes, and you need to track this information throughout the various phases of compilation. We suggest you read Section 4 of the Bryant/O'Hallaron note on *x86-64*

Machine-Level Programming available from the Resources page, especially the paragraph on move instructions and the effects of 32 bit operators in the upper 32 of the 64 bit registers.

Your code must strictly adhere to the struct alignment requirements explained above. You may also read the Section 3.1.2 in the *Application Binary Interface* description available from the Resources page.

Run-time Environment

The tests will be run in the standard Linux environment on the lab machines; the produced assembly code must conform to those standards. We recommend the use of `gcc -S` to produce assembly files from C sources which can provide template code and assembly language examples.

If your compiler detects any (compile-time) errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0.

7 Changes

Revision 1. The syntax of the `new` operator has been changed to require parentheses around its argument. This avoids nasty ambiguities such as `* * new int * * 5`.

Revision 2. The `void*` type has been removed from the source grammar. It is used purely internally. Moreover, subtyping is now defined only with reflexivity and $\mathbf{void}* \leq \tau*$. This simplified type-checking and avoids some non-sensical situations (which, however, did not seem to add any further unsoundness to the language).