15-411 Compiler Design: Lab 3
Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian and Nathan Snyder

Test Programs Due: 11:59pm, Tuesday, October 5, 2010
Compilers Due: 11:59pm, Thursday, October 14, 2010

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L3*. This language extends *L2* with the ability to define functions and call them. This means you will have to change all phases of the compiler from the second lab. One can write some interesting recursive and iterative functions over integers in this language. Correctness is still paramount, but performance starts to become a bit more of an issue as we may slightly tighten the time-bounds for the compiler.

2 Requirements

As for Lab 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 lexical specification of *L3* remains unchanged from that of *L2*. The syntax of *L3* is the superset of *L2* as presented in Figure 1. Ambiguities in this grammar are resolved according to the same rules of precedence as in *L2*.

⟨program⟩	::=	ϵ | ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩	::=	⟨fdecl⟩ | ⟨fdef⟩ | ⟨typedef⟩
⟨fdecl⟩	::=	⟨type⟩ **ident** ⟨param-list⟩ **;**
⟨fdef⟩	::=	⟨type⟩ **ident** ⟨param-list⟩ ⟨block⟩
⟨param⟩	::=	⟨type⟩ **ident**
⟨param-list-follow⟩	::=	ϵ | **,** ⟨param⟩ ⟨param-list-follow⟩
⟨param-list⟩	::=	**( )** | **(** ⟨param⟩ ⟨param-list-follow⟩ **)**
⟨typedef⟩	::=	**typedef** ⟨type⟩ **ident ;**
⟨type⟩	::=	**int** | **bool** | **ident**
⟨block⟩	::=	`{` ⟨decls⟩ ⟨stmts⟩ `}`
⟨decls⟩	::=	ϵ | ⟨decl⟩ ⟨decls⟩
⟨decl⟩	::=	⟨type⟩ **ident ;** | ⟨type⟩ **ident =** ⟨exp⟩ **;**
⟨stmts⟩	::=	ϵ | ⟨stmt⟩ ⟨stmts⟩
⟨stmt⟩	::=	⟨simp⟩ **;** | ⟨control⟩ | **;** | ⟨block⟩
⟨simp⟩	::=	**ident** ⟨asop⟩ ⟨exp⟩ | **ident** ⟨postop⟩ | ⟨exp⟩
⟨simpopt⟩	::=	ϵ | ⟨simp⟩
⟨esleopt⟩	::=	ϵ | **else** ⟨stmt⟩
⟨control⟩	::=	**if (** ⟨exp⟩ **)** ⟨stmt⟩ ⟨elseopt⟩ | **while (** ⟨exp⟩ **)** ⟨stmt⟩
	|	**for (** ⟨simpopt⟩ **;** ⟨exp⟩ **;** ⟨simpopt⟩ **)** ⟨stmt⟩
	|	**continue;** | **break;** | **return** ⟨exp⟩ **;**
⟨arg-list-follow⟩	::=	ϵ | **,** ⟨exp⟩ ⟨arg-list-follow⟩
⟨arg-list⟩	::=	**( )** | **(** ⟨exp⟩ ⟨arg-list-follow⟩ **)**
⟨exp⟩	::=	**(** ⟨exp⟩ **)** | ⟨intconst⟩ | **true** | **false** | **ident**
	|	⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ | **ident** ⟨arg-list⟩
⟨intconst⟩	::=	**num** (in the range $0 \leq$ **num** $< 2^{32}$)
⟨asop⟩	::=	`=` | `+=` | `-=` | `*=` | `/=` | `%=` | `&=` | `^=` | `|=` | `<<=` | `>>=`
⟨binop⟩	::=	`+` | `-` | `*` | `/` | `%` | `<` | `<=` | `>` | `>=` | `==` | `!=`
	|	`&&` | `||` | `&` | `^` | `|` | `<<` | `>>`
⟨unop⟩	::=	`!` | `~` | `-`
⟨postop⟩	::=	`++` | `--`

Non-terminals are in ⟨angle brackets⟩. Terminals are in **bold**. The absence of tokens is denoted by ϵ.

Figure 1: Grammar of *L3*

4 *L3* Elaboration

We need to provide static semantics for each of the `gdecls`:

- `fdecl`: which declares a function. This is very similar to function delcarations in C.
- `fdef`: which defines a function. Once again, very similar to C.
- `typedef`: which defines an alias for a type that is entirely transparent to the type system.

The semantics become more complicated because the language also intrinsically supports a foreign function interface through a mechanism of headers. Consult the section on the compile-time environment for details on how headers are supplied. For now, it is sufficient to know that a header is a list of `gdecls` that satisfy the following conditions:

- They are treated as if they are available to the *L3* program before or above the `gdecls` in the *L3* sources in a syntactic sense.
- They can contain `typedefs` or `fdecls`, but NOT `fdefs`. These declarations can be used to call functions that have been defined by the compilation environment.
- The fact that a `fdecl` was supplied in a header is an essential distinguishing factor that is preserved for the purpose of the static semantics.

To expeditiously capture the similarity and differences between gdecls in headers and gdecls in *L3* sources, we assume that the parser has parsed the header with the same rules as for sources, but has tagged the gdecls as external. On paper, we will represent this supposition by prefixing the production rules for gdecls with the dummy terminal `extern`.

Keeping to the precedent set by *L2* we isolate the dirty work of context sensitive syntax checking and desugaring in the elaborator. However, depending on how you prefer to organize your work, this section may very well be called "static semantics". The following elaboration strategy serves both as a specification and an implementation hint.

To capture the behaviour of gdecls, we recommend that you enhance the abstract syntax with the following

$$\begin{array}{ll}
\text{adecl} & ::= \text{nil} \mid \text{extfdecl}(f, \tau) \mid \text{intfdecl}(f, \tau) \mid \text{fun}(f, \tau, \text{params}, s) \\
\text{aprog} & ::= \text{adecl} \mid \text{aseq}(\text{adecl}, \text{aprog}) \\
\text{params} & ::= \epsilon \mid (x : \tau, \text{params}) \\
\tau & ::= \text{int} \mid \text{bool} \mid (\tau, \ldots, \tau) \supset \tau
\end{array}$$

s is a statement as defined in the abstract syntax of *L2*. Note that $(\tau, \ldots, \tau) \supset \tau$ is the abstract syntax for function types. The types in the parentheses are the types of the arguments, and the type at the right is the return type. There is no concrete syntax for function types, because the grammar does not allow functions to appear as expressions.

Elaboration, which was previously context insensitive, is now given under several contexts. Whole programs and gdecs are elaborated under three simultaneously maintained contexts – Δ is the set of external function identifiers, Ω is the context of type identifiers bound to the types that they alias, and Σ is the set of all defined functions identifiers. We begin by elaborating the top level of a program.

$$\frac{}{\langle \Delta; \Omega; \Sigma \rangle \vdash \epsilon \rightsquigarrow \text{nil}} \qquad \frac{\text{gdecl@}\langle \Delta; \Omega; \Sigma \rangle \rightsquigarrow \text{adecl}[\Delta'; \Omega'; \Sigma'] \quad \langle \Delta'; \Omega'; \Sigma' \rangle \vdash \langle\text{program}\rangle \rightsquigarrow \text{aprog}}{\langle \Delta; \Omega; \Sigma \rangle \vdash \langle\text{gdecl}\rangle \quad \langle\text{program}\rangle \rightsquigarrow \text{aseq}(\text{adecl}, \text{aprog})}$$

In defining the top-level elaboration of a program, we have captured that the elaboration of any gdecl adds information to the contexts, and that gdecls are elaborated in the given order to accumulate the information. We accomplish this by writing the rules to elaborate individual gdecls in a state-passing style. This is often a useful way to modularize rules, especially when multiple rules update the contexts in multiple ways. In our particular case, the judgement

$$\langle\text{gdecl}\rangle\text{@}\langle \Delta; \Omega; \Sigma \rangle \rightsquigarrow \text{adecl}[\Delta; \Omega; \Sigma]$$

means that we elaborate $\langle$gdecl$\rangle$ at some context, to produce an adecl and an updated context. The updates all preserve monotonicity of the context.

$$\frac{f \notin \text{Dom}(\Omega) \quad \Omega \vdash t \rightsquigarrow \tau \quad \Omega \vdash \langle\text{param-list}\rangle \rightsquigarrow \text{tlist}}{\textbf{extern}\ \text{t f}\ \textbf{(}\ \langle\text{param-list}\rangle\ \textbf{);}\text{@}\langle \Delta; \Omega; \Sigma \rangle \rightsquigarrow \text{extfdecl}(f, \text{tlist} \supset \tau)[\Delta, f; \Omega; \Sigma]}$$

$$\frac{f \notin \text{Dom}(\Omega) \quad \Omega \vdash t \rightsquigarrow \tau \quad \Omega \vdash \text{param-list} \rightsquigarrow \text{tlist}}{\text{t f}\ \textbf{(}\ \langle\text{param-list}\rangle\ \textbf{);}\text{@}\langle \Delta; \Omega; \Sigma \rangle \rightsquigarrow \text{intfdecl}(f, \text{tlist} \supset \tau)[\Delta; \Omega; \Sigma]}$$

In the aforementioned rules:

- Function names are not allowed to collide with type names.
- Elaboration of a parameter list is handled separately, under the context of type identifiers. You should check that the formal parameter names are unique and none of them are also visible typedef'd names.
- Every bit of elaboration that produces a type τ in its result occurs under the context Ω. This is in order to substitute abstract syntax types for every occurrence of typedef'd identifiers.

Here is how Ω is used to handle types and typedefs.

$$\frac{t' \notin \text{Dom}(\Omega) \quad \Omega \vdash t \rightsquigarrow \tau}{\textbf{typedef}\ \text{t t'}\ \textbf{;}\text{@}\langle \Delta; \Omega; \Sigma \rangle \rightsquigarrow \text{nil}[\Delta; \Omega, t' : \tau; \Sigma]}$$

External typedefs can be handled by an identical rule. Note that typedef'd names do not conflict with function names available before it. Observe that we discard type aliases in the elaborated program. We handle all namespacing issues at elaboration. This sort of erasure if often done in practice with transparent type aliases, because it is difficult to use them in large programs to produce meaningful type errors without also have some notion of saturating for every alias that you can give complex types. Where type aliases are preserved, there isn't a strong guarantee that type checking or type inference will use them when reporting type errors.

$$\frac{\tau \in \{\text{int, bool}\}}{\Omega \vdash \tau \rightsquigarrow \tau} \qquad \frac{\Omega(t) = \tau}{\Omega \vdash t \rightsquigarrow \tau}$$

Extend these rules to handle parameter lists where necessary. Finally, we get to function definitions.

$$\frac{f \notin \Delta \quad f \notin \mathrm{Dom}(\Omega) \quad f \notin \Sigma \quad \Omega \vdash t \rightsquigarrow \tau \quad \Omega \vdash \langle\text{param-list}\rangle \rightsquigarrow \text{tlist} \quad \Omega \vdash \langle\text{block}\rangle \rightsquigarrow \text{s}}{\text{t f } \mathbf{(}\ \langle\text{param-list}\rangle\ \mathbf{)}\ \langle\text{block}\rangle\ \mathbf{;}@\langle\Delta;\Omega;\Sigma\rangle \rightsquigarrow \text{fun}(f, \text{tlist} \supset \tau, params, s)[\Delta;\Omega;\Sigma, f]}$$

Note the following:

- As usual, we check for collision with type names. However, here we also check for collision with external declarations.
- The elaboration of blocks to statements is also done under the context of type aliases, unlike in *L2*. You should enhance the rules from *L2* to ensure that variable names within the body of a function don't collide with type names. You should also substitute types for type variables.
- There is no rule for `extern` function definitions. Headers should not contain function definitions.
- The design of the language is supposed to allow separate compilation in a manner similar to that present in C. Therefore, not every declared function is required to be defined. See the details of the compilation and runtime environment for further information. We do check that we don't define a function more than once.

5 *L3* Static Semantics

Type checking changes a bit.

- The typing rules for the entire program can be specified under a single context binding identifiers to types. Since functions types have an abstract syntax, they can also be added to the same context.
- You should add compatibility rules to check that all declarations and definitions of the same function name have compatible types, i.e. matching return types, the same number of arguments, and matching argument types.
- Function call expressions are similar to C. A function must be called with the correct number of arguments, and with compatible types. The whole expression has the corresponding return type.
- Function names are recursively bound. Therefore, when descending into a function's body to type check it, the function must be bound to its type in the context. Similarly, function parameters and their types should also be available when type checking the body. They are bound after the function name.
- As in *L2*, local variables are not allowed to shadow other local variables. However, local variables are allowed to shadow function names. This is similar to the behavior of C. You can easily emulate this behavior by enhancing the rules for declarations from *L2* as follows:

$$\frac{x \notin \mathrm{Dom}(\Gamma) \quad \Gamma, x : \tau \vdash s \textit{ valid}}{\Gamma \vdash \text{declare}(x, \tau, s) \textit{ valid}}$$

$$\frac{\Gamma(x) = (\tau_1, \ldots, \tau_n) \supset \tau' \quad \Gamma, x : \tau \vdash s \textit{ valid}}{\Gamma \vdash \text{declare}(x, \tau, s) \textit{ valid}}$$

- Note that for all purposes of type checking and shadowing, the parameters of a function have parity with local variables.

As you can probably observe, we have imported a lot of the non-uniform behavior of C to *L3*, especially with respect to name collisions and shadowing. It is plausible to handle these inconsistencies to varying extents in the elaborator and the type checker. You are free to make design decisions that suite your compiler. However, wherever you draw your module boundaries (if at all), think carefully about why your implementation is equivalent to this specification.

Control flow analysis is performed on each defined function as it was for `main` in *L2*. Note that the precise condition given for initialization checking in *L2* ensures that function parameters are not affected by the check. For all practical purposes, they can be deemed to be initialized at the beginning of the function.

Finally, since we are adding functions to the language, we no longer treat `main` as a keyword. It is an identifier with the special requirement that it can only be defined as a function with no parameters and the return type of `int`. Control flow in any *L3* program begins in the `main` function. Please refer to the section describing the runtime.

6 *L3* Dynamic Semantics

The dynamic semantics is extended directly from that of *L2*.

Function calls $f(e_1, \ldots, e_n)$ are very similar to their counterparts in C with the following significant difference: they must evaluate their arguments from left to right before passing the resulting values to f.

We also allow expressions to appear as statements, because we now have the concept of expressions with side-effects. These side-effects are quite simple in *L3*: divide by zero exceptions, and non-termination.

7 *L3* Compilation and Runtime Environment

Your compiler should accept a command line argument `-l` which must be given the name of a file as an argument. For instance, we will be calling your compiler using the following command: `bin/l3c -l l3rt.h test.l3`. Here, `l3rt.h` is the ubiquitous header mentioned in the sections on elaboration and the static semantics.

The gnu compiler and linker will be used to link your assembly to the implementations of the external functions, so you need not worry much about the details of calling to external functions. Just ensure that the code you generate adheres to the C ABI for linux on x86-64. Also recall the provision that all declared functions need not be defined. As long as you mangle all your function names, the linker should prevent the complete compilation of any test that does not define all used symbols.

Finally, we require that your compiler mangles all internally defined function names with the `_l3_` prefix sometime after static analysis and before code generation. We guarantee that the runtimes that we supply will not contain any symbols with an `_l*_` prefix. This is a simple hack to ensure that your test programs don't accidentally collide with functions that exist in our runtime, but do not get supplied with the header. Remember that the runtime will be calling `_l3_main` to execute the generated code.

8 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`. They are to be formatted and handed in as in *L2*.

You can also feel free to call any of the functions in `l3rt.h`. You may find them helpful for diagnostic purposes, and they will also test that you adhere to the calling convention correctly.

Infinitely recursing programs might either raise 11 or 14, depending on whether they first run out of memory or time. Test files should therefore only verify that *some* exception is raised.

Now that the language supports function calls, we would like some fraction of your test programs to compute "interesting" functions on specific values; please briefly describe such examples in a comment in the file. Disallowed are programs computing Fibonacci numbers, factorials, greatest common divisors, and minor variants thereof. Please use your imagination!

Compiler Files

The compiler sources are to be handed in as in *L2*. The compiler and the make target should be called `l3c`.

Using the Subversion Repository

Handin and handout of material is via the course subversion repository.

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab3` subdirectory. Or, if you have checked out `15411-f10/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform one of

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>/lab3/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>/lab3/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include any compiled files or binaries in the repository!

What to Turn In

Hand in on the Autolab server:

- At least 20 test cases, at least two of which generate as error and at least two others raise a runtime exception. The directory `tests/` should only contain your test files and be submitted via subversion or as a tar file as described above. 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. You may hand in as many times as you like before the deadline without penalty. 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 Tue Oct 5, 2010**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion or as a tar file as described above. 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. You may hand in as many times as you like before the deadline without penalty.

  Compilers are due **11:59pm on Thu Oct 14, 2010**.

9 Notes and Hints

Compiling Functions

It is not a strict requirement, but we recommend compiling functions completely independently from each other, taking care to respect the calling conventions but making no other assumptions. Interprocedural program analysis and optimization is difficult and, if you do it at all, is better left to a later lab.

Calling Conventions

Please refer to the course webpage for resources on the ABI and calling conventions. Note this oft-forgotten rule: `%rsp` must be 16-byte aligned. GCC often ignores this rule when it isn't actively using floats, because the requirement is only consequential when the floating point stack is in use.