Phobos: A front-end approach to extensible compilers

Adam Granicz and Jason Hickey
California Institute of Technology
1200 E. California Blvd, Pasadena, CA 91125, USA
{granicz,jyh}@cs.caltech.edu

Abstract

[1] *This paper describes a practical approach for implementing domain-specific languages with extensible compilers. Given a compiler with one or more front-end languages, we introduce the idea of a "generic" front-end that allows the syntactic and semantic specification of domain-specific languages. Phobos, our generic front-end, offers modular language specification, allowing the programmer to define new syntax and semantics incrementally.*

A key feature of our approach is the use of an open term language that can be used to describe arbitrary syntax, and the use of a term rewriting engine to encode semantic actions. The term language is expressive. Scoping can be defined explicitly, and term rewrites use second-order substitution, allowing the use of higher-order abstract syntax if needed.

Given a language specification and a source string, the generic front-end constructs a push-down automaton (PDA) based on the supplied grammar, lexes the source string, and simulates the constructed PDA with the stream of tokens obtained. During parsing, rewrite rules associated with grammar productions are executed, producing a single term when the PDA accepts. This term is then converted via further rewriting into a compiler representation and compilation proceeds to generate executable code.

1 Introduction

General-purpose programming languages offer numerous features that make them applicable to large domains of problems. But as software complexity increases, so does the number of problems that may appear due to the lack of higher-level formalisms. Naturally, such formalisms are difficult to obtain for the large problem spaces that general-purpose languages target. Instead, efforts should be concentrated through domain analysis [24] to restrict the problem domain at hand, so that a precise formalism can be found to guide development within that domain.

Domain-specific languages (DSL's) can provide a higher-level of abstraction in a notation that best suits the problem at hand. Often, domain-specific information can be used to perform optimization, or to pinpoint conceptual flaws in a solution. Using DSL's, larger and more complex domain-specific problems are easier to solve, requiring less effort to develop and maintain code and offering additional benefits in code reuse and modular design.

Rewriting systems have been studied extensively, and their relevance to parsing, which itself is a process of rewriting, is well known [18]. Furthermore, they can be used to express the operational and algebraic semantics of programming languages [14], and it is therefore natural to investigate the feasibility of connecting syntactic specification and formal semantic specification of programming languages.

The integration of a multilingual compiler and a theorem prover has great difficulties and even greater advantages. Indeed much research has concentrated on endowing theorem provers with ordinary computing abilities, or enabling formal verification of compiler components and source programs. Our research has focused on creating a formal compiler, called the Mojave Compiler Collection (MCC) [15], which makes extensive use of the MetaPRL [12] Logical Programming Environment (LPE). As a first step in their integration, we have developed a generic front-end we call Phobos to the Mojave compiler. Phobos provides a domain-specific language framework in which the programmer can write language specifications that define syntax, semantics, and optimizations based on domain-specific knowledge. These modules can be refined and extended through simple inheritance and added to the Mojave compiler dynamically, enabling users to compile any language for which such specification is available.

The rest of this paper is structured as follows. Section 2 discusses related work in syntax extension, formal language specification and tool generation, and extensible grammars and compilers. Sections 3, 4, 5 discuss the Mojave compiler architecture and its relevant components; Phobos and MetaPRL. In Section 6 we illustrate the design by implementing a small functional language using higher-order syntax. Finally, we finish with conclusions and future directions.

2 Related Work

Our work has strong ties with syntax extension, formal language specification, and extensible grammars and compilers.

Syntax extension for existing programming languages has been widely studied. Macro languages and preprocessors provide limited improvement of expressiveness by textual substitution, often ignoring important details such as variable capture, typing or scoping constraints. Other approaches involve abstract syntax or similar tree constructs [6], stream parsers [7], and typed macro systems [26]. In its most successful forms, syntax extension solves some of the challenges of specializing the syntax of a programming language for a particular problem domain, but provides no means for the incorporation of domain-specific semantics, often restricts the class of languages that can be extended, and typically involves unintuitive programming [6, 7]. Our representation uses a term language that can encode higher-order abstract syntax, and the underlying logical engine uses rewrite rules that avoid variable capture. Furthermore, we aim to decouple syntax and semantics specification, making it easy to incorporate domain-specific information.

Formal language specification and subsequent programming language tool generation are challenging and integral topics in the area of domain-specific languages. The first aims to provide formal descriptions of syntax and semantics, while the latter studies the efficient tool generation from such formal specifications. ASF+SDF [21], Gem-Mex [20], LISA [22], SmartTools [13] are some examples of recent systems available. These typically provide language editing capabilities and efficient language tool generation, including lexers and parsers, from language specifications similar to those of Phobos. Nevertheless, most do not allow for dynamic extension and integration but rather concentrate on separate tool generation and their off-line integration. This is accomplished by combining generic representations of specification artifacts, such as labeled trees and grammar tables, and the generated source code. For instance, LISA generates Java source files based on local templates, and compiles them at design time. Similarly, ASF+SDF translates rewriting specifications to C code. Phobos differs from these systems in that its primary goal is to allow for dynamic extension of the Mojave compiler without any intermediate source code and compilation. Phobos is a generic front-end, through which language syntax and semantics can be specified along with the target source files to be compiled.

The central problem in most rewriting-based tool generators and syntax extenders is that of scoping and introducing new bindings in semantic actions. For this reason, some of the systems mentioned are restricted to functional languages, have awkward binding predicates or simply offer no guarantees about proper typing. In Phobos, the term representation includes the binding structure, and rewriting rules automatically avoid variable capture in substitutions by alpha renaming. On the other hand, some systems are notably more mature and robust, and have been used in commercial applications. Many features deserve credit, such as ASF+SDF's list matching. A short survey of programming language tools and semantic specification can be found in [10].

Extensible grammars [19] form an essential component of similar systems. Phobos uses the model developed by Cardelli, Matthes, and Abadi, although at the moment our system only allows grammar addition and extension, but not update. On the other hand, the lexical specification is quite complete, and includes multiple regular expressions per terminal symbol, and allows inheriting language specifications to extend, remove, or override symbols. We allow the generation of illegal abstract syntax, which can be caught before conversion to compiler representation. Furthermore, our system allows arbitrary number of semantic action patterns per grammar production and the specification of LALR(1) grammars, whereas most extensible grammars are LL(1).

3 System Architecture

The Mojave compiler architecture is illustrated in Figure 1. It supports the Phobos extensible front-end as well as multiple fixed front-ends, each of which maps a general-purpose source language and its abstract syntax to a distinct intermediate representation (IR). These IR's are standardized low-level intermediate languages in continuation-passing style, designed for stepwise reduction of various front-end features to the common Mojave Functional IR (FIR). The FIR is an extension of System F [8], and constitutes the core of the compiler. We aim to exploit the formal properties of the FIR by integrating the MetaPRL formal system, through which FIR optimizations and other formal operations can be expressed, including program verification and type checking. Finally, Phobos allows the syntactic and semantic definition of new languages as an extension to one of the general-purpose front-end languages, or as a formal language defined directly in MetaPRL. These are illustrated as paths (i) and (ii), respectively, in Figure 1.

The entire system is written in the OCaml [25] program-

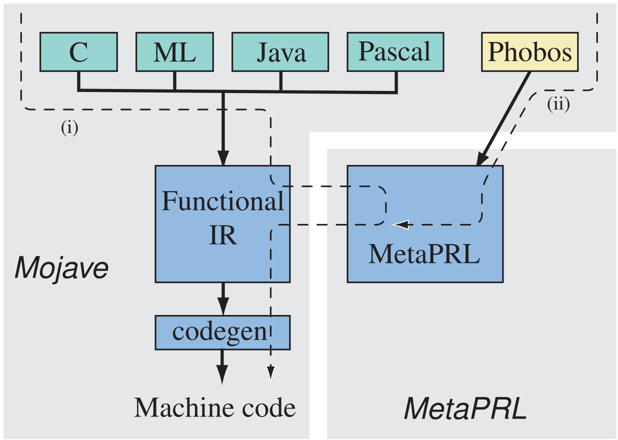

Figure 1. The Mojave compiler architecture

ming language. MetaPRL uses a term language to represent program syntax, while the Mojave compiler uses its own internal representations. The FIR⇆MetaPRL links are responsible for converting between the Mojave FIR and MetaPRL FIR representations [5]. Finally, the back-end generates object code, currently for the Intel x86 platform.

4 Phobos

Phobos provides generic scanning and parsing, and is designed to take advantage of the MetaPRL formal system for formal language development. Given a language definition and a source string, Phobos lexes and parses the source according to the syntax and semantic actions defined in the language. The result is a MetaPRL term that encodes the meaning of the source string with respect to the language specification provided.

Compiler options guide the process of further conversion of the resulting term. It may be converted to any of the abstract syntax representations supported by one of the fixed front-ends of the compiler, or may be passed to MetaPRL for FIR conversion. The rationale for the first is simple: we may want to use Phobos to extend one of the general-purpose languages, or we may want to simplify the task of implementing a new language by reusing an existing compilation path.

The Phobos→MetaPRL→FIR path requires the language designer to perform compilation using term rewriting, which for some languages can be quite difficult, since for many of the steps involved such as closure conversion, frame allocation and global optimizations, one needs global information to guide program rewriting. On the other hand, we are investigating performing such transformations with MetaPRL's inference rules, which provide contexts and conditional rewriting.

A typical Phobos language definition has these parts:

Module *module-name*
[**Include** *parent-modules*]*
[**Terms** -extend *module* { *term-declarations* }]*
[{ *global-rewrites* }]*
Tokens [-longest | -last] { *lexical-content* }
[%left | %right | %nonassoc *symbols*]*
Grammar -start *symbol* { *grammar-productions* }
[**Rewrites** { *post-parsing-rewrites* }]*

4.1 Language module inheritance

Phobos implements a simple scheme of inheritance between related language modules. An inheriting module imports all syntax and semantics implemented by the parent modules, with collisions resolved as follows:

- for global rewrite rules: new rules are added to those defined in parent modules. If two rewrite rules have the same redex, the latter one takes precedence.
- for terminal symbols: definitions can extend, remove, or override existing ones. In case of more than one matching definition, priority can be given the longest or the last matching symbol.
- for disambiguation rules: sets of rules defined in inheriting modules replace those found in the parent module.
- for grammar rules: the latest production rule (and its associated semantic actions) overrides any previous instances of the same rule.

Semantic analysis occurs through post-parsing rewrite rules. Inheriting language modules add new sets of rules to existing ones, and all sets are applied in the order of their definition.

Modules to be included are specified as strings. A single identifier can be used to refer to one of the internal modules, such as `Fc_ast` which declares all terms used in the C front-end's abstract syntax, or `Phobos_base` which defines basic arithmetic and relational operators, numbers, and numerous miscellaneous rewrite rules.

Include "module-name$_1$", ..., "module-name$_n$"

4.2 Term declarations

A MetaPRL term implicitly encodes the module that defined it, for scoping purposes. Similarly, in Phobos all occurrences of terms are rewritten to include their parent module, which is specified either directly (such as `Phobos_base!number`, where `Phobos_base` is the MetaPRL module) or through Phobos term declarations. For instance, in Figure 2, the terms `sum` and `prod` are declared as part of the MetaPRL module `Expression`. Thus, any occurrence of `sum` will be rewritten to `Expression!sum`. Furthermore, a special module designated as "@" is reserved for terms temporary in nature, such as terms encoding terminal symbols or those used as temporaries in multi-step term rewriting.

As good practice, terms should be declared with their parameter types and descriptive variable subterms (see next section on term syntax).

Terms -extend "Expression" {
    declare sum{'e1; 'e2}, prod{'e1; 'e2}...
}

Figure 2. Term declarations

4.3 Global rewrite rules

Global rewrites are used to define frequently-used term operations, and provide an elegant and concise notation by hiding rewriting details. Global rewrites are applied throughout the entire parsing process along with any syntax-related rewrite rules. The MetaPRL refiner applies all such rules from the top-most term repeatedly, until a fixpoint is reached.

Phobos extends the syntax for terms with a wildcard term (?) that stands for any term. This is used in rewrites where some of the subterms are irrelevant, as in $prod\{?; 0\} \rightarrow 0$.

4.4 Lexical specification

Terminals symbols are defined by a unique name and a set of regular expressions. In Figure 3, HEX extends NUM by adding a regular expression for hexadecimal numbers, while ANY may extend HEX with binary numbers and overrides all occurrences of NUM or HEX in the grammar that accompanies the lexical specification. On the other hand, if binary numbers should no longer be matched under ANY, one can remove them with the -remove "regexp" option. Note also that terminals not used in the abstract syntax can be discarded by placing a star symbol before their definition.

Before lexical analysis, Phobos constructs finite automata $\alpha_1, \alpha_2, \ldots, \alpha_n$ that accept the disjunction of regular expressions associated with each terminal. These automata then are used to partition the input string into a list of tokens. Each time a new token is found its source position is calculated, and along with the matched string and corresponding terminal name it is added to the list of tokens already discovered. If the end of the input string has not been reached and no symbol matches, a lexical error is reported at the current position. If more than one symbol matches, either the last in the order of definition or the longest match can be selected.

Tokens -longest {
    NUM = "[0-9]+" {
        __token__[p:**s**]{'pos} → number[p:**n**]
    }
    HEX -extend NUM = "0x[0-9a-fA-F]+" ...
    ANY -extend HEX -override NUM, HEX = ...
    ANY -remove "0b[0-1]+" ...
    ...
    * SPACE = " " {}
}

Figure 3. Lexical specification fragment

4.5 Syntax specification

Phobos allows the definition of ambiguous grammars, which are often more descriptive and natural [1], and it uses precedence and associativity information for disambiguation. Associativity can be defined for terminal symbols, and precedence can be derived from their ordering, much the same way as in YACC [17].

Syntax is defined using context-free grammars written in Backus-Naur Form (BNF), and each production is accompanied by a list of rewrite patterns. Phobos accepts LALR(1) languages, for which the standard parsing algorithm can be found in several sources, including Appel [4]. Figure 4 gives examples of grammar productions with one (which use the shorthand notation) or more rewrite rules.

The grammar definition is complete when one of the nonterminal symbols is declared to be the start symbol and specified as an option to the `Grammar` section. At this point, the actual start production is created with the specified nonterminal and EOF in its body.

Parsing is performed by a pushdown automaton (PDA) that is generated for each grammar. The PDA's stack contains *terms* that represent the symbols of the program. The language of terms is defined by MetaPRL, discussed in the next section. When the PDA shifts it pushes the current term

```
%left PLUS MINUS
%left TIMES DIV

Grammar -start exp {
    exp ::= NUM<'num> ⇒ 'num
    | ID<'id> ⇒ 'id
    | exp<'e1> PLUS exp<'e2> ⇒ sum{'e1; 'e2}
    | exp TIMES exp {
        'e1 TIMES 0 → 0
        | 0 TIMES 'e2 → 0
        | 'e1 TIMES 'e2 → times{'e1; 'e2}
        ...
    }   ...
}
```

Figure 4. Syntax specification fragment

onto its stack along with the current state. Similarly, when it reduces a grammar production the terms corresponding to the production's body are rewritten according to the first matching rewrite rule that is associated with the production. If there are no associated rewrite rules, the stack terms are combined into a default tuple term. In both cases the resulting term replaces the related terms on the stack.

During parsing, tokens obtained from the lexical analyzer are translated to special `__token__` terms which carry the matched string and source position, and may be rewritten if the corresponding terminal symbol was given a lexical rewrite rule.

For instance, if part of the input string was `1*2`, and numbers are rewritten to MetaPRL number terms, and binary operators are processed without rewriting, the parser's stack (its top being on the right) would contain

[...; 1; __token__["*"]{<some position>}; 2]

At this point, given the grammar in Figure 4, the parser will reduce the multiplication, and replace the top three terms on its stack with a new term.

[...; times{1; 2}]

If the parser encounters an unexpected symbol, it rejects and syntax error is reported at the current position. Furthermore, we report an error when no matching rewrite rule can be found when the parser is reducing a production. Otherwise, parsing proceeds until all tokens are consumed and the final derivation of the start symbol is obtained, at which point the parser accepts.

The rewriting system during parsing consists of the matching semantic rewrite rule associated with the current production and a possibly empty list of global rewrite rules. Rewriting at each step is done from the top-most term and until a fix-point is reached.

4.6 Conversion

Upon accepting, the resulting term may undergo further rewriting. Sets of rewrite rules can be applied consecutively before the final term is obtained. These rewrites can be used to perform simple optimizations, such as constant folding or dead-code elimination [5], and to convert to concrete terms, for instance to those found in front-end abstract syntax, or the FIR.

The Mojave C front-end supports the ANSI C standard with extensions for exceptions and polymorphism. Its abstract syntax is sufficiently general with high-level constructs such as for/while/do loops and exceptions, making it a good target for imperative languages. Its corresponding term set, defined in module `Fc_ast`, includes OCaml values such as `true{}` or `false{}`, and type constructors such as `option{None{}|Some{...}}` and `list{...}`. It also declares terms representing basic list operations, such as `list_create{...}`, `list_append{...}`, and `list_empty{}`.

Conversion into internal compiler representation is straightforward. Terms are matched by their names, and their subterms are interpreted according to their corresponding internal OCaml constructor. In case of a typing error an exception is raised, otherwise the term is successfully converted and passed back to the Mojave compiler.

5 MetaPRL

MetaPRL is a *logical framework*. The system architecture has three main parts, shown in Figure 5. The refiner implements the term rewriter; the meta-language is used to define term rewriting strategies; and the theories are Phobos language definitions. MetaPRL includes a general purpose theorem prover, which can be used to develop logics and program semantics for the languages we define. However, our interest here is in the term rewriting system, which is implemented in the refiner. The core of the refiner includes the term module, which defines term syntax and operations, and the logic engine, which defines term rewriting and theorem proving.

All program syntax is expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has an operator name, which is a unique name that identifies the language and operator of the term. For example, the operator name for addition in the C language is `Fc_ast!add`, where `Fc_ast` is the AST language definition for C, and `add` is the specific operator in the language.

Next, the term has an optional list of parameters representing constant values. The parameters are used to build

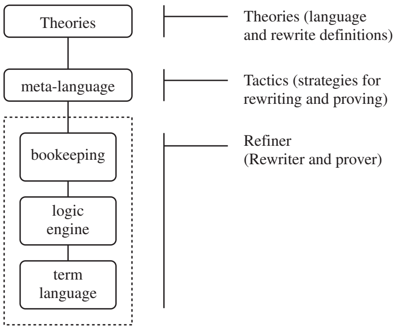

Figure 5. MetaPRL system architecture

the ground terms for terminal symbols, including numbers(n), strings(s), variables(v), etc. Finally, each term may have a list of subterms with possible variable bindings. We use the following syntax to describe terms, based on the NuPRL definition [3]:

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1; \cdots; p_n]}_{parameters}\underbrace{\{\vec{v_1}.t_1; \cdots; \vec{v_m}.t_m\}}_{subterms}$$

A few examples are shown at the right. Variables are terms with a variable parameter for their name; numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm $'b$. We often use the notation `'v` as a shorthand notation for variables. The single quote uniquely identifies the construction as a variable, although we will often omit the quote when it can be inferred from context.

Printed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda{x.'b}`
$f(a)$	`apply{'f; 'a}`
v	`var[v:v]{}`
$x+y$	`sum{'x; 'y}`

Term rewriting is defined as a set of rules, where each rule includes a pattern to be rewritten (a *redex*), and a value that is the result of the rewrite (the *contractum*). For example, the rule for beta-reduction in the untyped lambda calculus would be expressed using the following rule.

rewrite beta : $apply\{lambda\{v.e_1[v]\}; e_2\} \rightarrow e_1[e_2]$

This declaration defines a rewrite rule called `beta` that can be applied to a beta redex, performing the substitution. Note that the statement of the rewrite uses second-order substitution [2, 23]. The pattern $e_1[v]$ matches a term in which the variable v is allowed to be free, and the term $e_1[e_2]$ in the contractum constructs the term matched by e_1 with e_2 substituted for v.

One important property of the term rewriting system is that variable binding and scoping is explicit, and term rewriting avoids capture. We illustrate this with a small example. Suppose we want to augment the C programming language with an $\mathbf{iterate}(e_1)\{e_2\}$ construction that acts like a loop that executes expression e_2 with number of iterations specified by expression e_1. This is a straightforward macro expansion in terms of the `for` loop: we introduce a new iteration variable i and final index j to prevent e_1 from being evaluated more than once. The rewrite is defined as follows (we use the conventional syntax to simplify the notation):

iterate$(e_1)\{e_2\} \rightarrow$
{ **int** i, j; j = e_1; **for**(i = 0; i != j; i++) { e_2 } }

The key part of the term representation is that the declaration for the variables i and j in the expansion defines a binding occurrence for these variables. Thus if we expand the program fragment $\mathbf{iterate}(10)$\{ a[i] = b[i]; i++; \}, the rewriter will rename variables to avoid capture, producing a program like the following. This is a byproduct of the formal system; capture avoidance is necessary for consistency.

```
{
    int i2, j;
    j = 10;
    for(i2 = 0; i2 != j; i2++) {
        a[i] = b[i];
        i++;
    }
}
```

There are other benefits of using MetaPRL for the term rewriting engine.

- Since MetaPRL is a logical framework, language definitions can be coupled with a semantics that can be used to reason about, optimize, and transform programs.
- MetaPRL implements a broad set of primitives and data types, such as lists, sets, trees, functions, and other data structures. We can use these data structures transparently.
- MetaPRL has been heavily optimized; current implementations achieve several millions of rewrites per second [11] on an Intel 400MHz Pentium II.

6 Example

In this section, we outline the design of MLight, a small ML-like language that can be used for numerical calculations and evaluated at parse time. Our goal is to illustrate the use of higher-order abstract syntax and the full power of our system by describing complete program evaluation, which can serve as a basis for partial evaluation in real applications and an integral part of many related optimizations

		Definition
exp	$::=$	num
	$\mid$	var
	$\mid$	$\mathbf{true} \mid \mathbf{false}$
	$\mid$	$exp[+ \mid - \mid * \mid / \mid \% \mid < \mid <= \mid > \mid >=]exp$
	$\mid$	$\mathbf{if}\, exp\, \mathbf{then}\, exp\, \mathbf{else}\, exp$
	$\mid$	(exp)
	$\mid$	$id(exp*)$
let	$::=$	$\mathbf{let}\, var = exp\, \mathbf{in}\, let$
	$\mid$	$\mathbf{let}\, var(var*) = let\, \mathbf{in}\, let$
	$\mid$	$\mathbf{let\, rec}\, var(var*) = let\, \mathbf{in}\, let$
	$\mid$	exp
top_let	$::=$	$\mathbf{let}\, var = exp$
	$\mid$	$\mathbf{let}\, var(var*) = let$
	$\mid$	$\mathbf{let\, rec}\, var(var*) = let$
$prog$	$::=$	top_let+

Figure 6. MLight syntax

including constant folding, function inlining, and dead-code elimination. The syntax of MLight is defined in Figure 6.

Our task has three distinct steps: parsing, performing linear conversion, and evaluation. During parsing, we denote program variables with MetaPRL variables, although no binding is performed. Then we linearize the source program, and make the binding structure apparent. Evaluation proceeds by substituting expressions in place of variables, ultimately resulting in a constant.

6.1 Parsing

We begin by including the necessary functionality from `Phobos_base`, which defines numbers, Booleans, and other basic types; basic arithmetic and relational operators; list operations, and many miscellaneous rewrites. Next, the term language that will be used to encode MLight abstract syntax is declared.

```
Module MLight

Include Phobos_base

Terms -extend "MLight" {
    declare binop[s]{'e1; 'e2}
    ...
}
```

Terminal symbols include numbers and Boolean values, identifiers, keywords, and operators. Note that identifiers are rewritten to MetaPRL variables and numbers are rewritten to built-in numbers.

```
Tokens -longest {
    ID = "[_a-zA-Z][_a-zA-Z0-9]*" {
        __token__[p:s]{'pos} → 'p
    }
    NUM = "[0-9]+" {
        __token__[p:s]{'pos} → number[p:n]
    }...
}
```

Next, we define the grammar of the language. Below is the segment that defines expressions and programs. Note that the term resulting from the start production is wrapped in an evaluation term whose first subterm denotes the evaluation context, initially empty. The evaluation term will be responsible for evaluating the entire program. We also wrap the list of top-level let definitions in a `linear{}` term that will be reduced according to the linear conversion defined next. Furthermore, note that token precedence is not shown.

```
Grammar -start prog {
    prog ::= top_let_list<'list>
            ⇒ eval{nil{}; linear{'list}}
    ...
    top_let ::= LET ID<'var> EQ exp<'e>
            ⇒ let_var{'var; 'e}

        | LET ID<'f> LPAREN opt_id_list<'l>
        RPAREN EQ body<'body>
            ⇒ let_fun{'f; 'l; 'body}

        | LET REC ID<'f> LPAREN opt_id_list<'l>
        RPAREN EQ body<'body>
            ⇒ let_fun_rec{'f; 'l; 'body}

    ...
    exp ::= NUM {}
        | ID {}
        | TRUE {}
        | FALSE {}
        | exp<'e1> PLUS exp<'e2>
            ⇒ binop["+"]{'e1; 'e2}
    ...
        | IF exp<'c> THEN exp<'t> ELSE exp<'f>
            ⇒ if{'c; 't; 'f}
        | ID<'f> LPAREN opt_exp_list<'l> RPAREN
            ⇒ apply{'f; 'l}
        | LPAREN exp<'e> RPAREN ⇒ 'e
}
```

6.2 Linear conversion

After parsing, the main program is represented as a list of top-level let definitions. To impose the variable bindings defined in the source program, we need to linearize these definitions. The result is a single term in which subterms have the same bound variables (modulo alpha-renaming) as at their corresponding point in the source program.

MLight linear conversion (denoted as $\mathbf{C}$ below) is very similar to continuation-passing style (CPS) conversion, and can be defined for a list of terms $el_1 :: \mathbf{T}$ as follows.

$$\mathbf{C}([exp]) = exp$$
$$\mathbf{C}([\mathbf{let}\, v = e]) \rightarrow \mathbf{let}\, v = \mathbf{C}(e)$$
$$\mathbf{C}([\mathbf{let}\, f(\ldots) = e]) \rightarrow \mathbf{let}\, f(\ldots) = \mathbf{C}(e)$$
$$\mathbf{C}([\mathbf{let\, rec}\, f(\ldots) = e]) \rightarrow \mathbf{let\, rec}\, f(\ldots) = \mathbf{C}(e)$$

$$\mathbf{C}((\mathbf{let}\, v = e) :: \mathbf{T}) \rightarrow \mathbf{let}\, v = \mathbf{C}(e)\, \mathbf{in}\, \mathbf{C}(\mathbf{T})$$
$$\mathbf{C}((\mathbf{let}\, f(\ldots) = e) :: \mathbf{T}) \rightarrow \mathbf{let}\, f(\ldots) = \mathbf{C}(e)\, \mathbf{in}\, \mathbf{C}(\mathbf{T})$$
$$\mathbf{C}((\mathbf{let\, rec}\, f(\ldots) = e) :: \mathbf{T}) \rightarrow$$
$$\mathbf{let\, rec}\, f(\ldots) = \mathbf{C}(e)\, \mathbf{in}\, \mathbf{C}(\mathbf{T})$$

These translate directly to the following rewrites.

Rewrites {
    linear{cons{number[num:**n**]; nil{}}} →
        number[num:**n**]
    linear{cons{var[var:**v**]; nil{}}} → 'var
    linear{cons{true{}; nil{}}} → true{}
    linear{cons{false{}; nil{}}} → false{}
    linear{cons{binop[op:**s**]{'e1; 'e2}; nil{}}} →
        binop[op:**s**]{'e1; 'e2}
    linear{cons{if{'cond; 'e1; 'e2}; nil{}}} →
        if{'cond; 'e1; 'e2}
    linear{cons{apply{'f; 'params}; nil{}}} →
        apply{'f; 'params}

    linear{cons{let_var{var[v:**v**]; 'exp}; 't}} →
        letvar{'exp; v.linear{'t}}
    linear{cons{let_fun{var[f:**v**]; 'p; 'e}; 't}} →
        letfun{'f; 'p; linear{'e}; linear{'t}}
    linear{cons{let_fun_rec{var[f:**v**]; 'p; 'e}; 't}} →
        letfun_rec{'f; 'p; linear{'e}; linear{'t}}
}

6.3 Evaluation

For specifying evaluation rules, we maintain an environment that contains all the function definitions in the current scope. Each entry in the environment is a tuple $(f, rec, params, body)$, where rec indicates whether f is recursive or not (we denote this by either $\mathbf{rec}$ or $\overline{\mathbf{rec}}$).

$$\mathbf{E}(\Delta; number) \rightarrow number$$
$$\mathbf{E}(\Delta; \mathbf{true}) \rightarrow \mathbf{true}$$
$$\mathbf{E}(\Delta; \mathbf{false}) \rightarrow \mathbf{false}$$
$$\mathbf{E}(\Delta; e_1[binop]e_2) \rightarrow \mathbf{E}(\Delta; e_1)[binop]\, \mathbf{E}(\Delta; e_2)$$

$$\mathbf{E}(\Delta; \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3) \rightarrow$$
$$\mathbf{E}(\Delta; \mathbf{if}\, \mathbf{E}(\Delta; e_1)\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3)$$

$$\mathbf{E}(\Delta; \mathbf{if\, true\, then}\, e_2\, \mathbf{else}\, e_3) \rightarrow \mathbf{E}(\Delta; e_2)$$
$$\mathbf{E}(\Delta; \mathbf{if\, false\, then}\, e_2\, \mathbf{else}\, e_3) \rightarrow \mathbf{E}(\Delta; e_3)$$

$$\mathbf{E}((\Delta_1, ((f; \mathbf{rec}; p_1 \ldots p_n; e), \Delta_2)\, \mathbf{as}\, \Delta_f); f(e_1 \ldots e_n))$$
$$\rightarrow \mathbf{E}(\Delta_f; \mathbf{let}\, p_1 = e_1\, \mathbf{in} \ldots \mathbf{let}\, p_n = e_n\, \mathbf{in}\, e)$$

$$\mathbf{E}((\Delta_1, (f; \overline{\mathbf{rec}}; p_1 \ldots p_n; e), \Delta_2); f(e_1 \ldots e_n)) \rightarrow$$
$$\mathbf{E}(\Delta_2; \mathbf{let}\, p_1 = e_1\, \mathbf{in} \ldots \mathbf{let}\, p_n = e_n\, \mathbf{in}\, e)$$

$$\mathbf{E}(\Delta; \mathbf{let}\, v = e_1\, \mathbf{in}\, e_2) \rightarrow \mathbf{E}(\Delta; e_2[\tfrac{\mathbf{E}(\Delta; e_1)}{v}])$$

$$\mathbf{E}(\Delta; \mathbf{let}\, f(params) = e_1\, \mathbf{in}\, e_2) \rightarrow$$
$$\mathbf{E}(((f; \overline{\mathbf{rec}}; params; e_1), \Delta); e_2)$$

$$\mathbf{E}(\Delta; \mathbf{let\, rec}\, f(params) = e_1\, \mathbf{in}\, e_2) \rightarrow$$
$$\mathbf{E}(((f; \mathbf{rec}; params; e_1), \Delta); e_2)$$

The last top-level let is evaluated as follows.

$$\mathbf{E}(\Delta; \mathbf{let}\, v = e) \rightarrow \mathbf{E}(\Delta; e)$$

$$\mathbf{E}(\Delta; \mathbf{let}\, f(params) = e) \rightarrow \mathbf{E}(\Delta; e)$$

$$\mathbf{E}(\Delta; \mathbf{let\, rec}\, f(params) = e) \rightarrow$$
$$\mathbf{E}(((f; \mathbf{rec}; params; e), \Delta); e)$$

These formal evaluation rules can be implemented with rewrite rules in a straightforward manner. We begin with evaluating constants.

    eval{'env; number[num:**n**]} → number[num:**n**]
    eval{'env; true{}} → true{}
    eval{'env; false{}} → false{}

Binary operations are evaluated in two steps. First, both operands are evaluated and the entire operation is rewritten to a result term. In the second step, the operation is evaluated according to the subterms of the result term. For instance, if both subterms are `true{}` and the operation is equality, the result is the true term. Other arithmetic and relational operators accept number operands only, and are mapped to the corresponding `Phobos_base` operators.

    eval{'env; binop[op:**s**]{'e1; 'e2}} →
eval{'env; res[op:**s**]{eval{'env; 'e1}; eval{'env; 'e2}}}

eval{'env; res["+"]{number[n1:**n**]; number[n2:**n**]}} →
        sum[n1:**n**, n2:**n**]
    ...

eval{'env; res["="]{true{}; true{}}} → true{}
...

Evaluating if expressions is very similar. First, we evaluate the Boolean condition. If it is true, we evaluate the then-expression, otherwise the else-expression.

eval{'env; if{'cond; 'e1; 'e2}} →
    eval{'env; if_res{eval{'env; 'cond}; 'e1; 'e2}}
eval{'env; if_res{true{}; 'e1; 'e2}} →
    eval{'env; 'e1}
eval{'env; if_res{false{}; 'e1; 'e2}} →
    eval{'env; 'e2}

Function application is a bit more complex. First, we look up the function to be called in the evaluation environment, check for the right argument arity, assign arguments to formal parameters, and evaluate these assignments followed by the body of the function. Note that we do not bind function variables to their definition so that inlining of recursive functions does not capture (and thus causes to be alpha-renamed) the recursive function variable. Below, we use `inline_function` for simplicity.

eval{'env; apply{'f;'args}} →
    eval{'env; inline_function{'env; 'f; 'args}}

Evaluating let-forms is straightforward. Note that the binding variable is declared bound in the expression that follows it. `add_fun` and `add_fun_rec` adds a new function entry to the environment.

eval{'env; letvar{'exp; var.'rest['var]}} →
    eval{'env; 'rest[eval{'env; 'exp}]}
eval{'env; letfun{var[f:**v**]; 'params; 'e; 'rest}} →
eval{add_fun{'env; var[f:**v**]; 'params; 'e}; 'rest}

eval{'env; letfun_rec{var[f:**v**]; 'params; 'e; 'rest}} →
eval{add_fun_rec{'env; var[f:**v**]; 'params; 'e}; 'rest}

The last top-level function definition or assignment is what executes the program.

eval{'env; top_letvar{'var; 'exp}} →
    eval{'env; 'exp}
eval{'env; top_letfun{var[f:**v**]; 'params; 'e}} →
    eval{'env; 'e}
eval{'env; top_letfun_rec{var[f:**v**]; 'params; 'e}} →
eval{add_fun_rec{'env; var[f:**v**]; 'params; 'e}; 'e}

6.4 Well-formedness

During evaluation, if any variable is encountered it must be free, otherwise it would have been replaced when evaluating an earlier let form.

eval{'env; var[v:**v**]} →
    error["Free variable in program"]

Type-checking of MLight programs can be performed as an additional step after evaluation. Ultimately, all MLight expressions are mapped to constants and operations, so well-formedness of MLight expressions can be determined when we rewrite operations. As mentioned earlier, only operations with valid operands are evaluated, therefore if we encounter any unreduced terms after evaluation, we report an error message.

Rewrites {
    eval{'env; if_res{?; ?; ?}} →
        error["'if' must have Boolean condition"]
    eval{'env; res[op:**s**]{?; ?}} →
        error["invalid operands"]
}

Note that we could also raise error messages with source locations, but throughout our treatment of MLight we have ignored position information for simplicity reasons. Phobos provides convenient position handling, and this extension to MLight would be straightforward. Furthermore, the reader is referred to [15] for the complete definition of MLight.

7 Conclusion and Future Work

We have outlined our generic front-end Phobos, and demonstrated its use in formal language development with a simple functional language whose syntax and operational semantics was formally defined. We utilized the formal system to perform full program evaluation; an essential component of many advanced optimizations. In our earlier work [9], we presented the design of CLIP, an imperative language with strong resemblance to C and Pascal, and showed its definition as an extension to C. In Aydemir et. al. [5], we formalized the FIR language in Phobos and expressed *formal* optimizations in MetaPRL. Some of our current efforts concentrate on formal definition and optimization of UNITY-based languages. All of the above languages are defined as dynamic front-ends to the Mojave compiler.

In Phobos, we make significant use of the MetaPRL formal system. MetaPRL provides many features, including the language of terms, a capture-avoiding term rewriter, and a broad set of data structures. In future work, we plan to take advantage of the MetaPRL logical framework for program reasoning, transformation, and synthesis.

We do realize though that there are several limitations of our approach. Most importantly, no consideration is given to rewriting termination [16] or Church-Rosser properties. Furthermore, the language designer must be familiar with the Mojave compiler's internal representations and their corresponding term sets. Furthermore, in order to manage languages with multiple ancestors it would be useful to have hierarchical grammars and lexical specifications.

On the other hand, Phobos can reuse most of the Mojave architecture, including front-end and FIR optimizations. Its initial aim is to offer an open term language and conversion into any of the internal representations, and thus dynamic addition of source languages to the Mojave compiler. Currently, Phobos can convert into the C front-end's abstract syntax and the FIR. We had demonstrated successfully that converting imperative languages to this abstract syntax is viable, although using rewrite rules can be tedious to accomplish more involved transformations [9]. We have also investigated a more formal approach to language specification, using Phobos for parsing and employing MetaPRL's more advanced inference rules for transformations [5]. The results are promising, and currently we are working on Formal Integrated Design Environments (FIDE) that allow dynamic language specification with formal reasoning.

Phobos can also be used to serve as a link between application libraries and domain-specific languages. Functionality implemented within application libraries can be "imported" into new language specifications via front-end features for external function calls. For instance, one could define a DSL for scientific computations using high-precision arithmetic implemented as a C library that users must link against when compiling their source programs with Mojave.

Furthermore, we intend to use Phobos to define meta-languages that can express and optimize code segments written in different domain-specific languages. The main advantage to such meta-languages is the ability to express computation in a language that is closest to the programmer's intuition, and the seamless integration of such computations.

References

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975.

[2] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.

[3] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.

[4] A. W. Appel. *Modern compiler implementation in ML: basic techniques*. Cambridge University Press, 1997.

[5] B. Aydemir, A. Granicz, and J. Hickey. Formal Design Environments. *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2002. Appears in NASA technical report NASA/ CP-2002-211736.

[6] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 31–42. ACM Press, 2001.

[7] Daniel de Rauglaudre. Camlp4. http://caml.inria.fr/camlp4, 2002.

[8] J.-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et sonapplication a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–69. Springer-Verlag, NY, 1971.

[9] A. Granicz and J. Hickey. Phobos: A front-end approach to extensible compilers (long version). Technical Report caltechCSTR:2002.006, Computer Science Dept., California Institute of Technology, 2002.

[10] J. Heering and P. Klint. Semantics of programming languages: a tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, 2000.

[11] J. Hickey and A. Nogin. Fast tactic-based theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, August 2000.

[12] J. J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 2001.

[13] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools: A Generator of Interactive Environments Tools. In Reinhard Wilhelm, editor, *10th International Conference on Compiler Construction*, volume 2027, pages 355–360. Lecture Notes in Computer Science, Springer-Verlag, 2001.

[14] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[15] J. Hickey, J. D. Smith, A. Granicz, N. Gray, C. Tapus, and B. Aydemir. The Mojave Research Group Website. http://mojave.cs.caltech.edu.

[16] Jean-Pierre Jouannaud. Rewrite Proofs and Computations. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139, pages 173–218. Springer Verlag, 1995.

[17] S. C. Johnson and R. Sethi. Yacc: a parser generator. pages 347–374, 1990.

[18] J. v. Leeuwen, editor. *Handbook of Theoretical Computer Science, Vol. B.* Elsevier Science Publishers, 1990.

[19] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report SRC 121, Digital Equipment Corporation Systems Research Center, 21 Feb. 1994.

[20] M. Anlauff and P.W. Kutter and A. Pierantonio. Formal aspects and development environments for Montages. In M.P.A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*. Electronic Workshops in Computing, Springer/British Computer Society, 1997.

[21] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24:334–368, 2002.

[22] Mernik Marjan, Lenic Mitja, Avdicausevic Enis, and Zumer Viljem. *LISA: An Interactive Environment for Programming Language Development*, volume 2304. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[23] A. Nogin and J. Hickey. Sequent schema for derived rules. In *Theorem Proving in Higher-Order Logics (TPHOLs '02)*, 2002.

[24] R. Prieto-Díaz. Domain analysis: an introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[25] D. Rémy and J. Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[26] Steven E. Ganz and Amr Sabry and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 74–85. ACM Press, 2001.