Phobos: A front-end approach to extensible compilers (long version)

Adam Granicz and Jason Hickey

California Institute of Technology, Pasadena CA 91125, USA
{granicz,jyh}@cs.caltech.edu
http://mojave.cs.caltech.edu/

Caltech Technical Report CS-TR:2002.006

Abstract. This paper describes a practical approach for implementing certain types of 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.

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 possible 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 can be concentrated, through domain analysis [19], to restrict the problem domain at hand so that a precise formalism can be found to guide development within that domain.

Domain-specific languages (DSLs) 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 DSLs, larger, more complex domain-specific problems are easier to solve, requiring less effort to develop and maintain code, with 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 [17]. Furthermore, they can be used to express the operational and algebraic semantics of programming languages, and it is therefore natural to investigate the feasibility of connecting syntactic specification and formal semantic specification of programming languages.

Our research has focused on creating a formal compiler, called the Mojave compiler collection (MCC), which makes extensive use of the MetaPRL [15]

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 without having to recompile the Mojave compiler itself.

The rest of this paper is structured as follows. Section 2 outlines the Mojave compiler architecture and discusses its relevant components; Phobos and MetaPRL. In Section 3 we describe the language of Phobos, and in Section 4 we illustrate the design by implementing a small imperative language called CLIP. Finally, we finish with conclusions and future directions.

1.1 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 [11], and typed macro systems [22]. 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, 11].

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 [12], Centaur [9], Pan [8], PSG [7], CIGALE [23] and Synthesizer Generator [21] are examples of the many 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.

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 [13].

Extensible grammars [10] form an essential component of the above 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. Furthermore, we allow the generation of illegal abstract syntax, which can be caught during term conversion. At the same time, 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), although we regenerate the parsing tables for extended grammars from scratch.

2 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 mapping a general-purpose source language and its abstract syntax to a distinct intermediate representation (IR). The front-end IRs are then converted to the common Mojave "functional" IR. This FIR constitutes the middle-end of the compiler, with connection to the MetaPRL formal system, which is used to express FIR optimizations and other formal operations. Phobos allows the definition of domain-specific languages as an extension to one of the general-purpose languages, or as a formal language defined directly in MetaPRL.

The entire system is written in the OCaml [20] programming language. MetaPRL uses a term language to represent program syntax, while the Mojave compiler uses its own internal representation. The FIR$\leftrightarrows$MetaPRL links are responsible for converting between the Mojave FIR and MetaPRL FIR representations. Finally, the back-end generates object code, currently for the Intel x86 platform.

2.1 Phobos

Phobos is used for mapping arbitrary source languages into any of the internal compiler representations, and provides generic scanning, parsing and conversion. Given a language definition and a source string, it 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

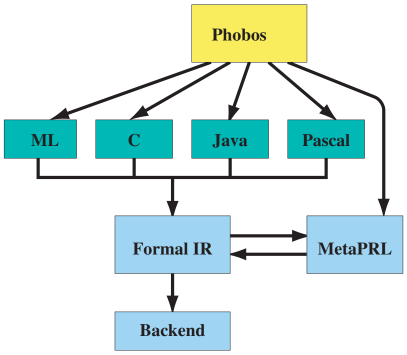

Fig. 1. The Mojave compiler architecture

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. In this paper, we aim to convert to one of the built-in abstract syntax representations, and leave the MetaPRL formal path to a future paper. For a Phobos implementation of the FIR language, and *formal* optimizations expressed in MetaPRL, the reader is referred to Aydemir et. al. [5].

Lexical analysis Given regular expressions $t_1, t_2, \ldots, t_n$, Phobos constructs finite automata $\alpha_1, \alpha_2, \ldots, \alpha_n$. These in turn are used to find the first or the longest matching substring s_i from position $p(i)$, where i is the number of iterations, $p(0) = 0$, and $p(n) = \sum_{i=0}^{n-1} p(i) + |s_n|$. If no matching substring can be found, a syntax error is reported at position $p(i)$.

Otherwise, 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.

Parsing Parsing is performed by a pushdown automaton (PDA) that is generated for each grammar. The PDA stack contains *terms* that represent the terminal and nonterminal symbols of the program. The language of terms is defined by MetaPRL, discussed in the next section. Semantic actions are defined by term rewrites that are applied to the terms on the stack when a production is reduced.

Phobos is an LALR(1) parser, for which the standard algorithm can be found in several sources, including [4]. Given the augmented start production

`<%start%>` ::= `<start> EOF`, the constructed PDA is simulated with the stream of tokens obtained from the generic scanner. The standard parsing algorithm is modified as follows. At any time the PDA can

– *shift a symbol* - the current token is converted into a special token term, and may be rewritten if its corresponding terminal symbol had a lexical rewrite rule. This term then is pushed onto the PDA's stack along with the current state.
– *reduce a production* - if the production has a list of rewrite rules, the first matching rule is located and the corresponding stack terms are rewritten accordingly. If there are no associated rewrite rules, the stack terms are combined into a default tuple term. In either case, the resulting term replaces the corresponding terms on the stack.
– *reject* - the source string is rejected, the position of the current token is reported as a syntax error.
– *accept* - source is syntactically valid; the only term on the stack, corresponding to the derivation of the start production, is passed for further processing.

The parsing rewriting system at any time consists of a matching semantic rewrite associated with the current production and a possibly empty list of global rewrite rules. Rewriting is applied until a fix-point is reached.

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 arithmetic simplification or dead-code elimination [5], and to convert to concrete terms, for instance to those found in front-end abstract syntax, or the FIR.

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

2.2 MetaPRL

MetaPRL is a *logical framework*. The system architecture has three main parts, shown in Figure 2. 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

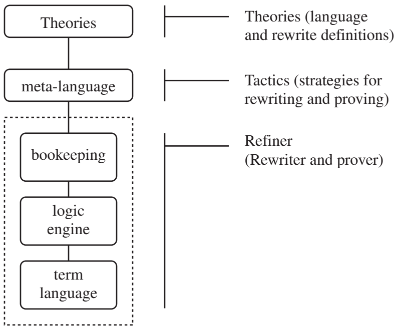

Fig. 2. MetaPRL system architecture

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 the ground terms for terminal symbols, including numbers, strings, 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]:

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

A few examples are shown at the right. Variables are terms with a string 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	`variable["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.

$$\textbf{rewrite}\ \text{beta}: \mathit{apply}\{\mathit{lambda}\{v.e_1[v]\}; e_2\} \longleftrightarrow e_1[e_2]$$

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, 18]. 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 **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):

$$\textbf{iterate}(e_1)\{e_2\} \longleftrightarrow \{\ \textbf{int}\ \mathrm{i, j; j} = e_1;\ \textbf{for}(\mathrm{i} = 0;\ \mathrm{i} \mathrel{!}= \mathrm{j};\ \mathrm{i}{+}{+})\ \{\ e_2\ \}\ \}$$

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 **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 [14] on an Intel 400MHz Pentium II.

3 Language specification with Phobos

A typical Phobos language definition has these parts:

Module *module-name*
Include *parent-modules* (*)
Terms -extend *module* { *term-declarations* } (*)

{ *global-rewrites* }
Tokens [-longest | -first] { *lexical-content* }
%left | %right | %nonassoc *symbols*
Grammar -start *symbol* { *grammar-productions* }
Rewrites { *post-parsing-rewrites* } (*)

The lines marked with a (*) may be omitted and/or repeated.

3.1 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 module, with collisions resolved as follows:

- for terminal symbols: both the new and old definitions are maintained and used, but priority can be given to either the longest match or the latter definition.
- for grammar rules: the latest production rule (and its associated semantic actions) overrides any previous instances of the same rule.
- for disambiguation rules: sets of rules defined in inheriting modules replace those found in the parent module.
- for global rewrite rules: new rules are added to those defined in parent modules.
- for post-parsing rewrite rules: sets of rules 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 the one declaring all terms used in the C front-end's abstract syntax (`Fc_ast`).

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

3.2 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 `Itt_int_base!number`, where `Itt_int_base` is the standard MetaPRL module for defining numbers) or through Phobos term declarations. For instance, in Figure 3, the terms `num`, `id`, `sum` ... `quot`, and `exp` are declared as part of the MetaPRL module `base_e`. 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.

```
Terms -extend "base_e" {
   declare num[n]{'pos}, id[s]{'pos}
   declare sum{'e1; 'e2}, ..., quot{'e1; 'e2}
   declare exp{'kind; 'pos}
}
```

Fig. 3. Term declarations

3.3 Global rewrite rules

Global rewrites are used to define frequently-used term operations. For instance, in Figure 4, `union_exp_pos` is used to create the union of two expression positions, given our term declaration earlier. This rewrite simplifies production of position information, which otherwise would have to be computed explicitly in each semantic action.

Global rewrites are applied throughout the entire parsing process along with any given semantic action. The MetaPRL refiner applies all these rewrites from the topmost term repeatedly, until a fix-point 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 the example below.

```
pos_of_exp{exp{?; 'pos}} -> 'pos
union_exp_pos{'e1; 'e2} -> union_pos{pos_of_exp{'e1}; pos_of_exp{'e2}}
```

Fig. 4. Global rewrites

3.4 Lexical specification

Terminals symbols are defined by a unique name and their corresponding regular expression. Each token, represented as a special `__token__` term that carries its matched string and source position, may be given lexical meaning by an optional lexical rewrite rule. Figure 5 shows a Phobos fragment that defines some of the lexical elements needed for simple arithmetic expressions.

If part of the input string can be matched by more than one regular expression, priority can be given to the first or the longest match with the corresponding option. Furthermore, terminal symbols not used in the abstract syntax can be excluded by placing a star symbol in front of their definition.

```
Tokens -longest {
   NUM = "[0-9]+"       { __token__[p:s]{'pos} -> num[p:s]{'pos} }
   ...
   * SPACE = " "        {}
}
```

Fig. 5. Lexical specification fragment

3.5 Syntax and semantics specification

Syntax is defined using context-free grammars written in BNF. Each production may be accompanied by a list of rewrite patterns that encode the corresponding semantic action.

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 [16].

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.

Figure 6 shows a fragment of grammar specification for simple arithmetic expressions. Note that the resulting *exp* term is encoded as `exp{'kind; 'pos}`, where `'kind` is the type of expression in our chosen term language, and `'pos` is the source position.

```
%left PLUS MINUS
%left TIMES DIV
%left LPAREN RPAREN

Grammar -start exp {
   exp ::= NUM          { num[p:s]{'pos} -> exp{num[p:s]; 'pos} }
      | ID              { ... }
      | exp PLUS exp    { 'e1 PLUS 'e2   -> exp{sum{'e1; 'e2};
                                               union_exp_pos{'e1; 'e2} }
      ...
}
```

Fig. 6. Syntax specification fragment

3.6 Term conversion

After parsing, further rewrite rules can be applied to perform simple optimizations and the conversion to an internal term set. One can control the rewrites that are executed by grouping them into sets that are executed together. These sets are augmented with the global rewrites, and similarly applied top-down until fix-point. For example, the rewrite in Figure 7 eliminates addition by zero.

```
Rewrites {
   sum{exp{num[n:s]; ?}; exp{num["0"]; ?}} -> num[n:s]
   ...
}
```

Fig. 7. Term conversion/optimization

4 Example

In this section, we will outline the design of a small imperative language called CLIP by extending the arithmetic expression module we developed in the previous section. Although the length of the entire language specification is roughly 200 lines, we include only parts of it here for brevity.

4.1 The Fc_ast term set

The Mojave C front-end supports the ANSI C standard with extensions for exceptions and polymorphism. Its abstract syntax (in Figure 8) is sufficiently general with high-level constructs such as `for/while/do` loops and exceptions, making it a good target for imperative languages.

The `Fc_ast` term set is a straightforward implementation of the C front end's abstract syntax (see Appendix). For instance, for the *expr* OCaml type in Figure 8, the `Fc_ast` term set defines a term in the form `expr{'kind; 'pos}`, where `'kind` can be `CharExpr{...}`, ..., `TypeDefs{...}`, and `'pos` is the built-in position term used in the lexical rewriting phase. Other AST-related OCaml types are similarly represented as terms. Integers are represented as `int[i:s]`, floats as `float[f:s]` and strings as `string[s:s]`.

The `Fc_ast` term set also includes OCaml values, such as `true{}` or `false{}`; and type modifiers, such as `option{None{}|Some{...}}` and `list{...}`. It also defines terms representing basic list operations, such as `list_create{...}`, `list_append{...}`, and `list_empty{}`, just to mention a few.

4.2 CLIP

CLIP (C-LIke Pascal) is a small imperative language with strong resemblance of Pascal and C. A short sample is given in Figure 9. CLIP has two built-in types: floating-point numbers and integers, and constructs for variable declarations, function definitions and applications.

	Constructor	Description
expr =	**UnitExpr**(pos, ...)	Unit value
|	**CharExpr**(*pos*,...)	Character constants
|	**IntExpr**(*pos*,...)	Integer constants
|	**FloatExpr**(*pos*,...)	Floating-point constants
|	**StringExpr**(*pos*,...)	Strings
|	**VarExpr**(*pos*,...)	Variables
|	**OpExpr**(*pos*,...)	Function/operator application
	...	Subscripting, projection, assignment, etc.
|	**IfExpr**(*pos*,...)	Conditional
|	**For/While/DoExpr**(*pos*,...)	For/While/Do-loop
|	**TryExpr**(*pos*,...)	Exception handling
|	**SwitchExpr**(*pos*,...)	Switch
	...	
|	**SeqExpr**(*pos*,...)	Sequencing
|	**VarDefs**(*pos*,...)	Variable declarations
|	**FunDef**(*pos*,...)	Function definitions
|	**TypeDefs**(*pos*,...)	Type definitions

Fig. 8. Mojave C abstract syntax

We start by including the language definition for arithmetic expressions, and declaring the additional terms we need. These include a term to represent floating-point numbers, descriptive terms for new terminal symbols, and terms that are used in conjunction with new global rewrite rules.

```
Module Clip

Include Fc_ast
Include "base_e.cph"

Terms -extend "@" {
   declare fnum[s]{'pos}
   declare VAR{'pos}
   declare FUNCTION{'pos}
   declare PROGRAM{'pos}
   declare INT{'pos}
   declare FLOAT{'pos}
   declare BEGIN{'pos}
   declare END{'pos}
   declare RETURN{'pos}

   declare COMMA{'pos}
   declare COLON{'pos}
   declare EQ{'pos}
   declare SEMI{'pos}
```

```
var global: integer = 2;

function square(a: integer): integer;
{
   return a*a;
};

program
{
   var i: integer = square(10);
   ...
   return i;
};
```

Fig. 9. A short CLIP program

```
   declare elist{'list; 'pos}
   declare binop[s]{'e1; 'e2}
   declare op[s]{'op_pos; 'list}
   declare ty_int{'pos}
   declare ty_float{'pos}
   declare main_ty{'pos}
   declare var_pattern[s]{'pos}
   declare var_patt_decl[s]{'pos; 'ty}
   declare main_params{'pos}
   declare exp_of_id{'id}
}
```

Next, we provide some helper rewrites. `binop[op:s]{'e1; 'e2}` takes a string parameter, the string representing the binary operator, and two subterms corresponding to the two operands. The contractum of the rewrite specifies how the same functionality is represented in the `Fc_ast` term set. Similarly, `op[op:s]{'op_pos; 'list}` takes a string parameter representing the function to be called, and two subterms, `'op_pos` representing the source position of the function application, and `'list` the parameters of the function. `ty_int{'pos}` and `ty_float{'pos}` return an `Fc_ast` integer and float type, respectively, parameterized by their source position. `main_ty{'pos}` is an alias for the return type of the main CLIP program, which we have defined to be integer. `var_pattern` and `var_patt_decl` are used to create variable patterns. `main_param{'pos}` constructs the parameters of a CLIP program, which are conveniently defined to be the same as for any regular C program. The last rewrite pattern constructs a variable expression from an identifier.

```
{
   binop[op:s]{'e1; 'e2} ->
      OpExpr{PreOp{};
             symbol[op:s]{union_exp_pos{e1; 'e2}};
             list{list_create2{'e1; 'e2}}}
```

```
   op[op:s]{'op_pos; 'list} ->
      OpExpr{PreOp{};
             symbol[op:s]{'op_pos};
             'list}

   ty_int{'pos} -> ty{TypeInt{StatusNormal{}; Int32{}; true{}}; 'pos}

   ty_float{'pos} -> ty{TypeFloat{StatusNormal{}; Double{}}; 'pos}

   main_ty{'pos} -> ty_int{'pos}

   var_pattern[v:s]{'pos} ->
      pattern{VarPattern{symbol[v:s]{'pos}; symbol[v:s]{'pos}; option{None{}}};
             'pos}

   var_patt_decl[v:s]{'pos; 'ty} ->
      patt_decl{'pos; var_pattern[v:s]{'pos}; 'ty}

   main_params{'pos} ->
      list{list_create2{var_patt_decl["argc"]{'pos; ty_int{'pos}};
         var_patt_decl["argv"]{'pos;
            ty{TypePointer{
               StatusNormal{};
               ty{TypePointer{
                  StatusNormal{};
                  ty{TypeChar{StatusNormal{}; Int32{}; true{}}; 'pos}}; 'pos}};
            'pos}}}}

   exp_of_id{id[v:s]{'pos}} -> exp{VarExpr[v:s]{}; 'pos}
}

Tokens -longest {
   FNUM = "[0-9]+\\.[0-9]+" { __token__[p:s]{'pos} -> fnum[p:s]{'pos} }
   VAR = "var"              { __token__[p:s]{'pos} -> VAR{'pos} }
   FUNCTION = "function"    { __token__[p:s]{'pos} -> FUNCTION{'pos} }
   PROGRAM = "program"      { __token__[p:s]{'pos} -> PROGRAM{'pos} }
   INT = "integer"          { __token__[p:s]{'pos} -> INT{'pos} }
   FLOAT = "real"           { __token__[p:s]{'pos} -> FLOAT{'pos} }
   BEGIN = "{"              { __token__[p:s]{'pos} -> BEGIN{'pos} }
   END = "}"                { __token__[p:s]{'pos} -> END{'pos} }
   RETURN = "return"        { __token__[p:s]{'pos} -> RETURN{'pos} }

   COMMA = ","              { __token__[p:s]{'pos} -> COMMA{'pos} }
   COLON = ":"              { __token__[p:s]{'pos} -> COLON{'pos} }
   EQ = "="                 { __token__[param:s]{'pos} -> EQ{'pos} }
   SEMI = ";"               { __token__[param:s]{'pos} -> SEMI{'pos} }

   * COMMENT = "//[^\\n]*"  {}
}
```

The associativity and precedence rules are as follows:

```
%right EQ
%left PLUS MINUS
```

```
%left TIMES DIV
%left LPAREN RPAREN
```

Now onto the grammar definition. First we extend the definition of *exp* by three additional productions for floating-point constants, assignment and function application. We also add productions for possibly empty lists of expressions.

```
opt_exp_list ::= _ /* empty */ => list{empty_list{}}
   | exp_list {}

exp_list ::= exp_list COMMA exp
      {  'list 'COMMA 'exp ->
            list{list_append{'list; 'exp}}
      }
   | exp
      {  'exp ->
            list{list_create{'exp}}
      }

exp ::= FNUM
      { fnum[p:s]{'pos} -> exp{fnum[p:s]; 'pos} }
   | exp EQ exp
      {  'exp1 'EQ 'exp2 ->
            exp{binop["="]{'exp1; 'exp2}; union_exp_pos{'exp1; 'exp2}}
      }
   | ID LPAREN opt_exp_list RPAREN %prec prec_apply
      {  'fun LPAREN{'pos1} list{'l} RPAREN{'pos2} ->
            exp{
               op["()"]{union_pos{'pos1; 'pos2};
                  list{list_insert{list{list_insert{list{'l}; exp_of_id{'fun}}};
                       exp_of_id{'fun}}}};
               union_pos{'pos1; 'pos2}}
      }
```

Next, we define productions for optional parameter lists, in the forms of $param_1 : type_1; param_2 : type_2; \ldots ; param_n : type_n$. The only allowed types are integers and floats.

```
opt_params ::= _ /* empty */ => list{empty_list{}}
   | params {}

params ::= params SEMI param
      {  'list 'SEMI 'param ->
            list{list_append{'list; 'param}}
      }
   | param
      {  'param ->
            list{list_create{'param}}
      }

param ::= ID COLON type
      {  id[id:s]{'pos1} 'COLON 'ty ->
            patt_decl{'pos1; var_pattern[id:s]{'pos1}; 'ty}
      }
```

```
type ::= INT
      { INT{'pos} ->
         ty_int{'pos}
      }
   | FLOAT
      { FLOAT{'pos} ->
         ty_float{'pos}
      }
```

We now are ready for declarations and definitions. Note that the main program is preceded by the `program` keyword.

```
def ::= VAR ID COLON type EQ exp SEMI
      {  VAR{'pos1} id[id:s]{'id_pos} 'COLON 'ty 'EQ 'exp SEMI{'pos2} ->
            exp{
               VarDefs{
                  list{list_create{var_decl{
                     'id_pos;
                     StoreAuto{};
                     var_pattern[id:s]{'id_pos};
                     'ty;
                     InitExpr{pos_of_exp{'exp}; 'exp}}}}};
               union_pos{'pos1; 'pos2}}
      }
   | FUNCTION ID LPAREN opt_params RPAREN COLON type SEMI body SEMI
      {  FUNCTION{'pos1} id[id:s]{'pos2} 'LPAREN 'params 'RPAREN
         'COLON 'ty 'SEMI 'body SEMI{'pos7} ->
            exp{
               FunDef{StoreAuto{};
                  symbol[id:s]{'pos2};
                  'params;
                  'ty;
                  exp{SeqExpr{'body}; union_pos{'pos1; 'pos7}}};
               union_pos{'pos1; 'pos7}}
      }
   | PROGRAM body SEMI
      {  PROGRAM{'pos1} 'body SEMI{'pos2} ->
            exp{
               FunDef{StoreAuto{};
                  symbol["main"]{'pos1};
                  main_params{'pos1};
                  main_ty{'pos1};
                  exp{SeqExpr{'body}; union_pos{'pos1; 'pos2}}};
               union_pos{'pos1; 'pos2}}
      }

body ::= BEGIN stmts END
      {  'BEGIN elist{'list; 'pos} 'END ->
            'list
      }
```

Finally, we define statements that are either `return` statements, simple expressions, definitions or declarations. Any CLIP programs are a list of these statements.

```
main ::= stmts
      {  elist{'list; 'pos} ->
            exp{SeqExpr{'list}; 'pos}
      }

stmts ::= stmts stmt
      {  elist{'list; 'pos1} exp{'e; 'pos2} ->
            elist{list{list_append{'list; exp{'e; 'pos2}}};
                  union_pos{'pos1; 'pos2}}
      }
   | stmt
      { exp{'e; 'pos} ->
            elist{list{list_create{exp{'e; 'pos}}}; 'pos}
      }

stmt ::= exp SEMI
      {  'exp 'SEMI ->
            'exp
      }
   | def {}
   | RETURN exp SEMI
      {  RETURN{'pos1} 'exp SEMI{'pos2} ->
            exp{ReturnExpr{'exp}; union_pos{'pos1; 'pos2}}
      }
```

The final grammar we developed contains 32 productions, and consists of roughly 200 lines of actual Phobos code. All we have left is to rewrite our remaining "abstract" syntax to actual `Fc_ast` terms.

```
Rewrites {
   id[s:s] -> VarExpr[s:s]{}
   num[i:n] -> IntExpr{rawint{Int32{}; true{}; int[i:s]{}}}
   fnum[f:s] -> FloatExpr{rawfloat{Double{}; float[f:s]}}

   sum{'e1; 'e2}  -> binop["+"]{'e1; 'e2}
   diff{'e1; 'e2} -> binop["-"]{'e1; 'e2}
   prod{'e1; 'e2} -> binop["*"]{'e1; 'e2}
   quot{'e1; 'e2} -> binop["/"]{'e1; 'e2}
}

Rewrites {
   exp{'kind; 'pos} -> expr{'kind; 'pos}
}
```

5 Conclusion and Future Work

We have outlined our generic front-end Phobos, and demonstrated its use with a simple imperative language. Our approach to extensible formal language specification is simple yet sufficient to accomplish dynamic extendability of our compiler, which was the main focus of our presented work.

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 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. 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 are the ability to express computation in a language that is closest to the programmer's intuition and offers the desired features, and the successful integration of such computations.

A The Mojave C abstract syntax

```
open Symbol
open Rawint
open Rawfloat

(*
 * Labels are external names.
 *)
type label = symbol
type var = symbol

type pos = string * int * int * int * int
```

```
type op_class =
   PreOp
 | PostOp

type ty =
   TypeUnit    of pos * type_status * int
 | TypePoly    of pos * type_status
 | TypeChar    of pos * type_status * int_precision * int_signed
 | TypeInt     of pos * type_status * int_precision * int_signed
 | TypeFloat   of pos * type_status * float_precision
 | TypeArray   of pos * type_status * ty * expr * expr
 | TypeConfArray
               of pos * type_status * ty * symbol * symbol * ty
 | TypePointer of pos * type_status * ty
 | TypeRef     of pos * type_status * ty
 | TypeProduct of pos * type_status * ty list
 | TypeEnum    of pos * type_status * enum_decl list label_option
 | TypeUEnum   of pos * type_status * (label * union_enum_decl list) label_option
 | TypeStruct  of pos * type_status * field_decl list label_option
 | TypeUnion   of pos * type_status * field_decl list label_option
 | TypeFun     of pos * type_status * ty list * ty
 | TypeVar     of pos * type_status * symbol
 | TypeLambda  of pos * type_status * symbol list * ty
 | TypeApply   of pos * type_status * symbol * ty list
 | TypeElide   of pos

(*
 * Record fields have an optional field width (in bits)
 *)
and enum_decl  = pos * symbol * expr option

and union_enum_decl = pos * symbol * ty option

and field_decl = pos * symbol * ty * expr option

(*
 * Patterns.
 *)
and pattern =
   CharPattern    of pos * rawint
 | IntPattern     of pos * rawint
 | FloatPattern   of pos * rawfloat
 | StringPattern  of pos * precision * int array
 | VarPattern     of pos * symbol * label * ty option
 | StructPattern  of pos * (symbol option * pattern) list
 | EnumPattern    of pos * symbol * pattern
 | AsPattern      of pos * pattern * pattern

(*
 * Expressions.
 *)
and expr =
   (* Base expressions *)
   UnitExpr       of pos * int * int
```

```
 | CharExpr       of pos * rawint
 | IntExpr        of pos * rawint
 | FloatExpr      of pos * rawfloat
 | StringExpr     of pos * precision * int array
 | VarExpr        of pos * symbol * symbol

   (* Functions and operators *)
 | OpExpr         of pos * op_class * var * label * expr list
 | ProjectExpr    of pos * expr * var
 | SizeofExpr     of pos * expr
 | SizeofType     of pos * ty
 | CastExpr       of pos * ty * expr

   (* Structured expressions *)
 | IfExpr         of pos * expr * expr * expr option
 | ForExpr        of pos * expr * expr * expr * expr
 | WhileExpr      of pos * expr * expr
 | DoExpr         of pos * expr * expr
 | TryExpr        of pos * expr * (pattern * expr list) list * expr option
 | SwitchExpr     of pos * expr * (pattern * expr list) list
 | LabelExpr      of pos * symbol
 | CaseExpr       of pos * pattern
 | DefaultExpr    of pos
 | ReturnExpr     of pos * expr
 | RaiseExpr      of pos * expr
 | BreakExpr      of pos
 | ContinueExpr   of pos
 | GotoExpr       of pos * symbol

   (* Composition *)
 | SeqExpr        of pos * expr list

   (* Definitions *)
 | VarDefs        of pos * var_decl list
 | FunDef         of pos * storage_class * symbol * label
                   * patt_decl list * ty * expr
 | TypeDefs       of pos * type_decl list

(*
 * Initializers.
 *)
and init_expr =
   InitNone
 | InitExpr of pos * expr
 | InitArray of pos * (symbol option * init_expr) list

(*
 * A variable declaration.
 *)
and var_decl = pos * storage_class * pattern * ty * init_expr

(*
 * Formal parameter for a function or
 * a variable declaration.
```

```
 *)
and patt_decl = pos * pattern * ty

and type_decl = pos * symbol * symbol * ty

(*
 * A program is a list of declarations.
 * The empty program is valid.
 *)
type prog = expr list
```

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. Stuart 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. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.
4. Andrew 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. Submitted.
6. Jonthan Bachrach and Keith 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. Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.
8. Robert A. Ballance, Susan L. Graham, and Michael L. Van de Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):95–127, 1992.
9. P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, 1988.
10. L. Cardelli, F. Matthes, and M. Abadi. Extensible Grammars for Language Specialization. In *Proceedings of the Fourth International Workshop on Database Programming Languages*, August 1993.
11. Daniel de Rauglaudre. Camlp4. http://caml.inria.fr/camlp4, 2002.
12. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF (reference manual). *ACM SIGPLAN Notices*, 24(11):43–75, 1989.
13. Jan Heering and Paul Klint. Semantics of programming languages: a tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, 2000.
14. Jason Hickey and Alexey Nogin. Fast tactic-based theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, August 2000.

15. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 2001.
16. Stephen C. Johnson and Ravi Sethi. Yacc: a parser generator. pages 347–374, 1990.
17. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Vol. B*. Elsevier Science Publishers, 1990.
18. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In *Theorem Proving in Higher-Order Logics (TPHOLs '02)*, 2002.
19. Rubén Prieto-Díaz. Domain analysis: an introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.
20. Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.
21. Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: a system for constructing language-based editors*. Springer-Verlag New York, Inc., 1989.
22. 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.
23. F Voisin. Cigale: A tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7(1):61–86, 1986.