Teaching Programming Languages through Language Implementation

Kent D. Lee
Computer Science
Luther College
leekentd@luther.edu

Abstract

To teach effectively we must connect what we teach to a student's past experience. A programming languages course typically introduces students to other ways of thinking about programming. For students to learn they must understand the importance of what they are learning. In addition, research on teaching suggests that for new information to be remembered it must be connected to prior experience in some way.

This paper presents a series of exercises providing a general outline for teaching a programming languages course. The exercises focus on implementing parts of compilers and/or interpreters for small languages. While implementing a language, students become familiar with many aspects of programming languages concepts. Using language implementation as a motivational goal, these topics can be tied together to form a cohesive course.

Introduction

This paper describes a series of exercises and content for the Programming Language Foundations course at Luther College. To understand the context of this paper, it is important to understand what type of students take this course at Luther. The typical student at Luther comes into this class having taken three computer science courses: Introduction to Programming (using Java), Data Structures (again using Java), and the Computer Architecture class where they get an introduction to C and C++ as well as a little assembly language. Students taking the Programming Language Foundations class have a solid background in object-oriented/imperative programming.

Some of the comments in this paper will reference a paper by Thomas Anthony Angelo [1] titled, A "Teacher's Dozen", which is a paper on good educational teaching practices. Many of his comments are supported by research which is referenced in his paper. The class is patterned around the principles in that paper. As experiences and projects are related in this paper, Angelo's paper will be referenced as appropriate.

Angelo's first point is that "Active learning is more effective than passive learning". Students must *do* and not just *see* to really understand a concept. This can be emphasized by giving assignments. However, lectures can also include exercises that may be just as effective as homework and sometimes more effective than lecturing. The series of exercises presented in this paper are an excerpt from a workbook provided to students at the beginning of Programming Languages at Luther.

The rest of this paper will follow the flow of the Programming Languages class at Luther, describing exercises and projects that support the concepts students learn in the class. The class takes a two-pronged approach. Students study two new language paradigms, functional and logic programming, while using these languages to implement other languages. The choice of functional language is somewhat optional, but ML seems a logical choice because it has a lot of support for language description including a scanner and parser generator. That, among other things, makes it a nice functional language to use. The only real choice of logic programming language is Prolog.

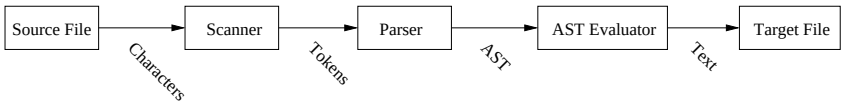

Figure 1: The structure of a compiler/interpreter

To study language implementation, students must be taught methods for describing syntax and semantics of languages. The following sections coincide with the parts of a compiler presented in figure 1. Interpreters don't generate target files, but the other parts in figure 1 could apply to interpreters as well.

In addition to the structure of a compiler, if students are to learn about language implementation they must also learn about models of computation. Stack machines and register machines together with the environments needed for language compilation, interpretation, and execution are explored in the exercises in the class.

The materials used to teach this class are from the workbook materials developed by the author and from two supplementary texts: 'Elements of ML Programming" by Jeffrey D. Ullman [14]. and the classic Prolog text, "Programming in Prolog" by Clocksin and Mellish [4]. Students end this class with two very good references to the languages they have learned and they use the books to get an introduction to the languages and as a reference for the languages in later projects.

```
1.<ramprog> ::= <executable> <equates> EOF
2.
3.<executable> ::= <labeled instruction> | <labeled instruction> <executable>
4.
5.<labeled instruction> ::= Identifier ":" <labeled instruction> | <instr>
6.
7.<instr> ::=
8.       <memref> ":=" Integer
9.   |   <memref> ":=" "PC" "+" Integer
10.  |   "PC" ":=" <memref>
11.  |   <memref> ":=" <memref>
12.  |   <memref> ":=" <memref> "+" <memref>
13.  |   <memref> ":=" <memref> "-" <memref>
14.  |   <memref> ":=" <memref> "*" <memref>
15.  |   <memref> ":=" <memref> "/" <memref>
16.  |   <memref> ":=" <memref> "%" <memref>
17.  |   <memref> ":=" "M" "[" <memref> "+" Integer "]"
18.  |   "M" "[" <memref> "+" Integer "]" ":=" <memref>
19.  |   "read" "("<memref> ")"
20.  |   "writeln" "(" <memref> ")"
21.  |   "goto" Integer
22.  |   "goto" Identifier
23.  |   "if" <memref> <condition> <memref> "then" "goto" Integer
24.  |   "if" <memref> <condition> <memref> "then" "goto" Identifier
25.  |   "halt"
26.  |   "break"
27.
28.<equates> ::= null | "equ" Identifier "M" "[" Integer "]" <equates>
29.
30.<memref> ::= "M" "[" Integer "]" | Identifier
31.
32.<condition> ::= ">=" | ">" | "<=" | "<" | "=" | "<>"
```

Figure 2: The BNF for the EWE language

Learning Syntax

To learn the syntax of a language it is common to be given a grammar describing the syntactically allowable programs of a language. Because students will be implementing small interpreters and compilers, they are first introduced to a low-level language. By doing this it provides a starting point to talk about models of computation. One such model is the von-Neumann architecture. Sethi [12] describes a language for a simple von-Neumann machine called RAM in his programming languages book. RAM stands for Random Access Machine. EWE is an extended and slightly modified version of this language that is suitable as a target language for the simple compilers implemented in this class. EWE doesn't stand for anything and should not be confused with μ-code! The BNF for the EWE language is given in figure 2. EWE is easy enough to learn and the syntax of the language should be familiar to most students since it is very Pascal-like. Because it is similar to languages most students are familiar with, it is a pretty easy transition for them to begin programming in EWE.

To emphasize how grammars can be used, students are presented with the grammar in figure 2 and asked to do several exercises relating to the it. For instance, during a lecture on grammars they are

asked to use the grammar in figure 2 to rewrite the following program so it is a valid EWE program with the same meaning as the original.

```
readln(A);
readln(B);
if A-B < 0 then
    writeln(A)
else
    writeln(B);
```

Another exercise asks the students to write a EWE program to read a number from the keyboard and print out the sum of all the numbers from 1 to the number. Most students will write a program with a loop to compute this. Finally, one important exercise has students read a list of numbers until zero is entered and then print the list of numbers in reverse order. This forces them to implement a stack in the EWE language which will be useful later when they map a stack machine into a von-Neumann architecture.

Language implementation issues concerning syntax include learning about scanners and parsers. Scanners, which are basically finite state machines based on regular expressions, are generally taught in a Discrete Structures course or possibly a Theory of Computation course. For these reasons, and in the interest of time, students in the Programming Languages course are provided with a scanner or a tool for making a scanner in each of the projects.

A parser is based on a grammar. Students are generally unfamiliar with grammars and several days are spent covering them. There may be some overlap with other courses, but rather than emphasize context-free vs context-sensitive grammars and the whole Chomsky Hierarchy, students are taught a working definition of grammars so they can practice deriving sentences. Students learn to do a left-most derivation and then are told how left-most derivations are constructed by top-down parsers.

Angelo's fourth principle of teaching says

> To be remembered, new information must be meaningfully connected to prior knowledge, and it must be remembered in order to be learned.

In the context of this class, students need to have a point of reference to work from. The first project assigned reinforces the working definition of a grammar that is covered in the class. This project is assigned in a language they are familiar with to help them connect with those past experiences. In the case of Luther students, their past experience has been with the Java language. This first project can come in several forms, including

- Construct a preorder expression interpreter
- Construct an inorder expression interpreter
- Construct an inorder to postorder or preorder converter

Each of these projects involves scanning input and converting the characters to tokens which can be done using the StreamTokenizer class in Java. However, this class is not easy to use so students at Luther use a Scanner class written by the author. The Scanner class has a getToken function and a putBackToken function. The preorder expression interpreter is the easiest to implement

```
public interface ASTNode {
  public double evaluate();
}

public class AddNode implements ASTNode {
  private ASTNode left, right;

  public AddNode(ASTNode l, ASTNode r) {
    left = l;
    right = r;
  }

  public double evaluate() {
    return left.evaluate() + right.evaluate();
  }
}
```

Figure 3: Java classes for an expression AST

because no look ahead is needed in the input to determine the next rule to apply. For a more challenging project one of the second two can be chosen, however it is important that students are successful early in the class without experiencing a lot of frustration.

The projects are written to parse the input using a hand written top-down parser. The parser produces an abstract syntax tree. These projects are perfect for reinforcing the concept of interfaces in Java, one of the most important features of the Java language. The AST (i.e. abstract syntax tree) produced by the parser can be implemented as a set of classes, one for each type of node in the AST. To evaluate an abstract syntax tree, each node in the tree should have an evaluate method. The code in figure 3 demonstrates how these classes and associated interface can be defined. The parser then operates by constructing the AST as it parses the input. If the third project above is selected the evaluate function can be replaced with a print function.

Learning to Program Functionally

After students have had some time to digest grammars and parsers while using a language they know, it's time to introduce them to the world of functional programming. Students who have learned imperative programming first need to unlearn many of those concepts, specifically variables and loops. It's not uncommon for the typical student to need some time struggling with recursion and the idea of writing programs that look more like a definition than code.

Students at Luther are assigned several exercises out of Ullman's book [14]. A series of three lectures introduce students to the concepts taught in the first four chapters of Ullman's book. Exercises are selected from the book and class time is used to work on the exercises so they can ask questions immediately. This cuts down on office hours while giving students a chance for immediate feedback. Working through these chapters takes about two weeks.

When students finish working the exercises in the book they are getting comfortable writing recursive functions. To implement languages in ML, students must know about ML datatypes. This is covered in the book and in a lecture on data structures in ML. At this point students are ready to write a compiler of expressions in ML. The compiler's structure is presented in figure 1.

Case Study: Implementing an Expression Interpreter

A key to success in any project is to make sure students have some direction so they aren't wandering aimlessly in the wilderness. Without some guidance, students may spend an inordinate amount of time in wasted exercises. This can have at least a couple of serious consequences. A problem that is too complex will frustrate students and they will become less motivated. In the words of Angelo [1]

> Learning is more effective and efficient when learners have explicit, reasonable, positive goals, and when their goals fit will with the teacher's goals.

The following pages present part of an interpreter of infix expressions written in ML. Students are motivated to learn how this interpreter works when they are told that the compiler they must implement will be based on this code with a few small changes. The big hurdle in this project is to get students thinking about the difference between compile-time and run-time issues. They likely have never struggled with this concept before and it will make most students think hard before they understand how to complete the compiler project. After completing the exercises in chapters one through four of Ullman's book and the additional exercises they get from the workbook, they have all the necessary background needed to understand the code presented here.

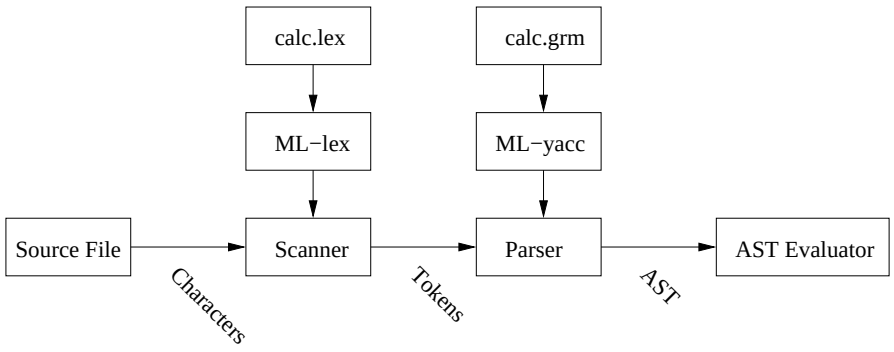

Figure 4: The structure of the expression intepreter project

```
structure calcAS =
    struct
        datatype
            Expr = add' of Expr * Expr
                 | sub' of Expr * Expr
                 | prod' of Expr * Expr
                 | div' of Expr * Expr
                 | negate' of Expr
                 | store' of Expr
                 | recall'
                 | integer' of int;
    end;
```

Figure 5: An ML datatype defining Abstract Syntax for Expressions

Some of the code for the interpreter is presented in this paper for reference. The structure of the interpreter project is given in figure 4. ML-lex and ML-yacc use the files *calc.lex* and *calc.grm* respectively to generate a scanner and parser. Given an expression, the parser returns an AST

```
SP:=100
one:=1
A := 5
M[SP+0] := A    # push the integer on the stack
SP := SP + one
A := 6
M[SP+0] := A    # push the integer on the stack
SP := SP + one
SP := SP - one
A := M[SP+0]    # pop one operand from the stack
SP := SP - one
B := M[SP+0]    # pop another operand from the stack
A := A + B      # add the two
M[SP+0] := A    # and push the result
SP := SP + one
writeln(M[100]) # print the final result
halt
equ SP M[0]
equ one M[1]
equ A M[2]
equ B M[3]
```

Figure 6: EWE code to simulate stack evaluation of addition

representing it. The AST defintion is found in figure 5. Note that values can be stored in a single global memory location using a postfix S operator to store a value, represented by the *store'* in the AST in figure 5. The memory location can be recalled using the R operator. For instance, 5S+R would yield 10.

The code in figure 12 begins by calling the interpret function which calls the parser which calls the scanner to read the input. The scanner reads the characters of the expression and translates them to tokens. The tokens are used by the parser to construct an AST. The AST evaluator takes an AST and evaluates it. The following sections give most of the contents for each of the files used in this project.

The Calculator Compiler

To implement an inorder expression compiler for the EWE target language it is helpful for students to understand what the compiler produces. One thing to decide upon is a model of computation. For expression evaluation, a stack is a logical choice of computation model. Students can write their compiler so it produces a EWE program that evaluates the expression on a stack.

Consider the expression 5+6. If it was evaluated on a stack machine the code would look something like this.

```
push 5
push 6
add   (* add should pop twice, add the numbers, and push the result *)
print (* pop the top of the stack and print it *)
```

```
fun compile filename  =
    let val (prog, _) = calcparse filename
         val eweFile = TextIO.openOut("a.ewe")
    in
         TextIO.output(eweFile,"\tSP:=100\n");
         TextIO.output(eweFile,"\tone:=1\n");
         codegen(eweFile,prog);
         TextIO.output(eweFile,"\twriteln(M[100])"^
                               "\t# print the final result\n");
         TextIO.output(eweFile,"\thalt\n");
         TextIO.output(eweFile,"\tequ SP M[0]\n");
         TextIO.output(eweFile,"\tequ one M[1]\n");
         TextIO.output(eweFile,"\tequ A M[2]\n");
         TextIO.output(eweFile,"\tequ B M[3]\n");
         TextIO.closeOut(eweFile)
    end
```

Figure 7: The compile function for the Calculator Compiler

In the interest of keeping students involved they can be asked to translate the pseudo-code above into a EWE program that simulates a stack. To give the students some guidance, it might be useful to tell them to define *SP* as a stack pointer in the EWE program, *one* as a constant for 1, and two memory locations to hold the operands for the addition operation. The two operands are called A and B in the solution presented in figure 6.

The Goal of this Project

The parser returns an AST representing the expression that was entered. Given the expression 5+6, the parser returns an AST as an ML datatype that looks like this

```
add'(integer'(5),integer'(6))
```

So, the goal is now to write a function that given a AST like the one above, prints the code in figure 6. To accomplish this goal, students should change the code in the file calc.sml of the project. They can replace the interpret function with a compile function like the one shown in figure 7.

The compile function in figure 7 calls the codegen function. The codegen function replaces the evaluate function in the original calc.sml. One more small change is needed. The run function in the calc.sml file should call the compile function instead of the original interpret function. That's it! Students need to write the compile and codegen functions to complete this project. While this project may look a little oversimplified, students get some satisfaction from seeing their compiler work and it does get them to understand the distinction between compile-time and run-time issues.

Extending the Compiler Calculator

The calculator compiler project can be used as the basis for several other projects concerning language implementation issues. One such issue is register allocation. While register allocation can be a complex topic due to its adhoc nature, Pittman and Peters [11] claim a simple demand-based

$AST_0 \rightarrow$ Prog AST
AST_1.min = 0
AST_0.val = AST_1.val
$AST_0 \rightarrow$ Add AST_1 AST_2 | Subtract AST_1 AST_2 | Multiply AST_1 AST_2 | Divide AST_1 AST_2
AST_1.min = AST_0.min
AST_2.min = AST_1.mout
AST_0.mout = AST_2.mout
AST_0.val = AST_1.val op AST_2.val where op is +,-,*,/ respectively
$AST_0 \rightarrow$ Store AST_1
AST_1.min = AST_0.min
AST_0.mout = AST_1.val
AST_0.val = AST_1.val
$AST_0 \rightarrow$ Recall
AST_0.val = AST_0.min
AST_0.mout = AST_0.min
$AST_0 \rightarrow$ number
AST_0.mout = AST_0.min
AST_0.val = number

Figure 8: An Attribute Grammar for the Expression Language

algorithm performs acceptably. A simple framework for register allocation is presented in [7]. Included with this framework is the ability to allocate registers by simulating pushing and popping a stack. Students can easily use this framework to convert their compiler to target a register machine instead of a stack machine. Since EWE can just as easily simulate a register machine as a stack machine, there is no need for the students to learn a new target architecture.

A programming languages class often includes a discussion of local bindings in languages. The ML language includes a *let ... in ... end* construct that is used to introduce local bindings. This construct can easily be introduced to the expression language. This project is nice because students get to modify the scanner, parser, and code generation function to complete the program. Furthermore, the addition of local bindings can be used later to introduce the compilation of functions in block structured languages. When local bindings are introduced, the S and R operators of the expression language could be dropped.

Attribute Grammars

There are many semantic methods that have been developed for language description. The key in an undergraduate course is presenting students with methods that are accessible to them and can keep their interest. Given that students know what a grammar is, attribute grammars are a good first choice for an example of a formal method. The expression language with store and recall operations, whose abstract syntax is given in figure 5, is a good example language for an attribute grammar. The existence of the global memory location can be modelled by a pair of attributes that single thread the memory location through abstract syntax trees of expressions. This provides the instructor with an example of both synthesized and inherited attributes.

The attribute grammar for the expression language is given in figure 8. Students can be asked to use the attribute grammar to decorate an abstract syntax tree for a given expression. This exercise and the example attribute grammar will be useful in a project presented in the next section.

Programming Logically

After introducing Prolog and working through appropriate exercises from the first three chapters of Clocksin and Mellish [4], students should be ready to discover how grammars can be written in Prolog. By showing students how grammars can be constructed in Prolog using difference lists they become confortable with the idea of creating a top-down parser using the grammar capabilities built into Prolog.

Students can then be asked to construct a grammar in Prolog for prefix expressions. In this assignment they are to construct a Prolog grammar for prefix expressions with the addition of a single memory location (like the grammar they previously constructed, but for Prolog this time). The grammar should contain parameters that build an abstract syntax tree of the expression. Then, they should write Prolog rules that evaluate the abstract syntax tree to get the resulting expression.

To complete this project students will want to use the readln predicate. Students can use the readln predicate as follows.

```
? - readln(L,_,_,_,lowercase).
```

When they do this they will get a prompt and then can enter some text. For instance, they might enter

```
|: + S 5 R
L = [+, s, 5, r] ;
No
?-
```

The readln predicate can only be satisfied once and reads one line of input. This will turn out to do nearly everything the students need. However, to make things easier while parsing, students can preprocess the list so that an expression like "+ S 5 R", which readln returns as [+,s,5,r], will look like [+,s,num(5),r] after preprocessing. The num tag on numbers will help them when they write their parsers. This assignment is very closely related to the attribute grammar given in figure 8. Students may wish to go back and review the notes on attribute grammars now that they know something about Prolog.

Action Semantics

While many formal methods of language description exist, most of the methods, due to their dense mathematical notation, are not readily accessible to the average undergraduate student. One exception to that is action semantics. While action semantics is a formal language, because of its English-like structure it can be understood on many different levels. While this section doesn't present any programming projects related to action semantics, it is a good example with which to wrap up the loose ends in a programming languages course. If taught the way this paper suggests, the overall theme of the course has been language implementation. That is also one of the primary goals of many formal methods for language description.

Formal semantic descriptions are generally not used in developing compilers that are in use today. Instead, compiler writers usually rely on English or some other natural language to give the

meaning of a programming language. While the English language is well-suited for conversation, its ambiguity causes problems in the defininition of programming languages. For example, because the semantics of Pascal was defined informally, the first two compilers for the language treated type equivalence differently: one implemented structural equivalence while the other expected name equivalence, even though Niklaus Wirth himself over-saw the construction of both compilers [15]. Mistakes like these may be unnecessary if research into formal methods of semantic description can eventually produce efficient compilers.

In action semantics, an *action* describes the meaning of a source language program. An *action semantic description* provides a mapping from the syntax of a programming language to the actions that describe its programs. In theory it is possible to automatically generate a compiler for a language given this mapping. Action semantics-based compilers and compiler generators are being studied by a small group of people. Research in this area began with the development of action semantics by Peter Mosses [8] and David Watt [15]. Action semantics-based compiler generation has been studied by Palsberg [10], Bondorf and Palsberg [2], Doh and Schmidt [5], Ørbæk [9], Brown, Moura, and Watt [3], and by Lee [6].

Actions describe the manipulation of one of three entities, collectively called the current information consisting of transient values, bindings of values to locations (i.e. cells), and bindings of identifiers to data. Accordingly, there are three facets to action semantics, the functional, imperative, and declarative facets, respectively. The current information and actions combine to describe one more model of computation. Genesis supports enough action notation to describe programming languages containing functions of one or more parameters, iteration, selection, sequential execution, and simple and mutual recursion.

Primitive actions in action semantics include actions like give 4, which is part of the functional facet of action semantics, giving a transient value. Transients are a tuple of data representing intermediate values used in computations. The action bind "x" to 4 comes under the declarative facet and creates a new binding. The action store 4 in the given cell imperatively stores a value in a cell.

Every action is either a *primitive action*, like those above, or a *compound action*, composed of other primitive and compound actions that are combined through the use of *combinators*. Combinators dictate how the current information is propagated to their sub-actions and how the sub-actions affect the current information. Most combinators are binary, combining two sub-actions into one compound action. For instance, consider the action

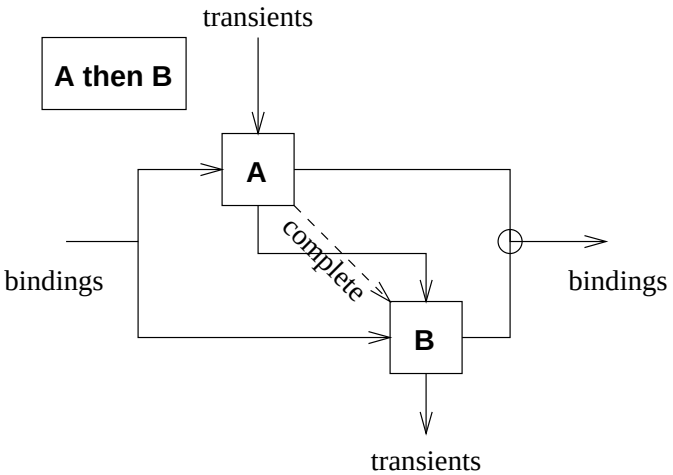

Figure 9: The then Diagram

```
| give 4
then
| bind "x" to the given value
```

In the functional facet the then combinator propagates the transients given by the first sub-action to the transients used by the second sub-action. So, in this example, (4) is given as the *outgoing* transient of the first sub-action and the second sub-action uses the singleton tuple (4) as its *incoming* transient value. The behavior of the then combinator is described by figure 9, where transients flow from top to bottom. Combinator diagrams were first introduced by Watt[15] and were revised by Slonneger [13].

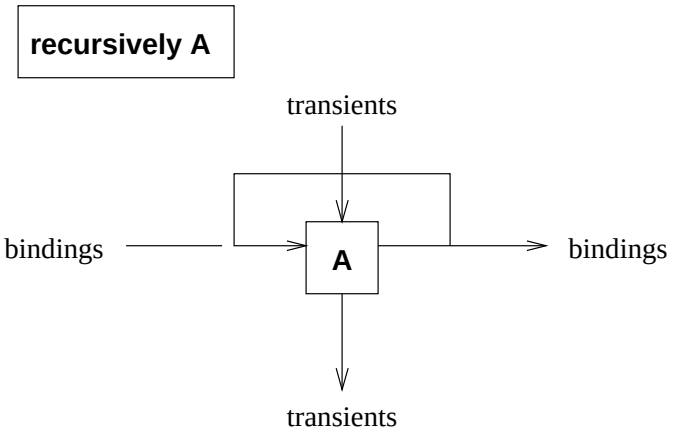

Figure 10: The recursively Diagram

Other combinators may affect other facets. For instance, recursively, depicted in figure 10, is an example of a unary combinator that affects the declarative facet. This combinator allows abstractions (i.e. functions) to be declared that are mutually recursive. The recursively combinator was first presented in [6].

Action notation includes support for sorts (i.e. types) like integer, truth-value, cell, and abstraction, and tuple. Other sorts may be added to action notation by specifying their semantics in terms of an algebraic specification. For a complete description of action notation refer to [8].

Figure 11 illustrates the structure of Genesis. Genesis begins by translating a *source program* to its *program action*. The program action is then checked to make sure it is consistent in terms of its sorts, or types. Since action's are high-level descriptions of computation, the program action must undergo transformations to simplify it so it may be easily translated into a low-level machine language. The action transformation occurs just before code generation.

Genesis is freely available and serves as a nice tool for demonstrating action semantics and how it can be used. Genesis can be freely downloaded from www.cs.luther.edu/~leekent. It requires the Bash shell, the make facility, and Standard ML to run. The package includes two sample action semantic descriptions, one for a small subset of ML called *Small* and the other for a small Pascal-like language. It also is capable of generating compilers that target either the Java Virtual Machine or a MIPS architecture. A MIPS simulator that accepts the MIPS code that is generated by Genesis is also available upon request.

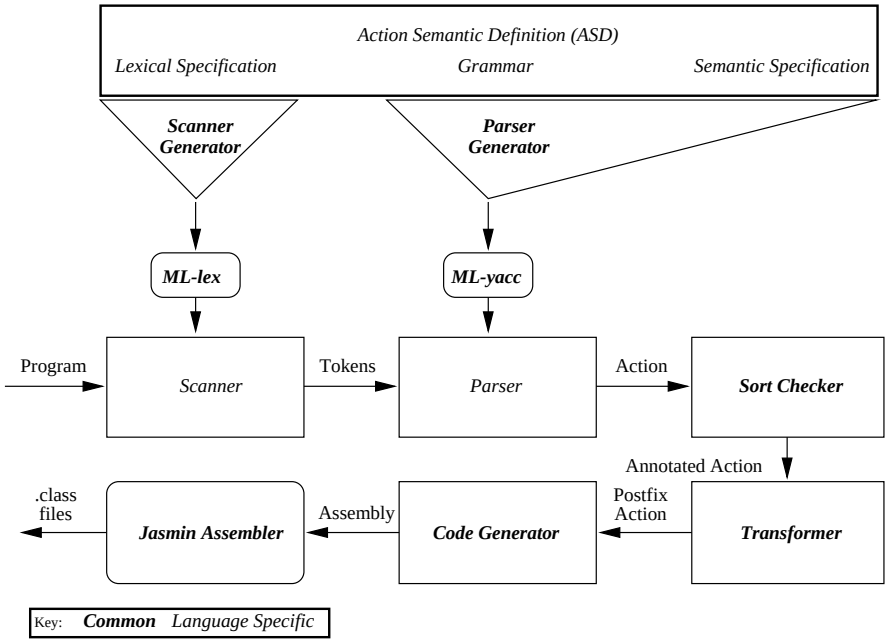

Figure 11: Structure of Genesis

Conclusion

This paper advocates the study of language implementation as a unifying goal in the study of programming language foundations. The study of interpreters and compilers allows teachers to discuss syntax and semantics of languages. At the same time, the goal of implementing these languages gives students a purpose in studying language implementation issues and concepts.

For students who are most familiar with the imperative style of programming, the study of functional and logic languages is a new experience that takes some time to adjust to. With the right level of instructional support, students can quickly begin implementing simple languages in using Standard ML and Prolog.

Models of computation can be a recurring theme in the course. By studying stack evaluation, register machines, and the model of computation provided by action semantics, students begin to learn the underlying structure of their programs.

Instructors are encouraged to consider studying action semantics as a formal method of language description. Action semantics is readily accessible to undergraduate students because of its English-like structure. However, it can be understood at deeper levels given the time to explore it in more detail. In addition, tools are available for action semantics to let students experiment with generating actions of their own.

In addition to the content of the course, this paper advocates a style of teaching where students are actively involved in many lectures by completing short exercises that coincide with the instructor's goals. This style of *in class participation* is supported by research on teaching [1].

References

[1] T.A. Angelo. A teacher's dozen. *AAHE Bulletin*, pages 3–13, April 1993.

[2] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FCPA '93)*, Copenhagen, DK, 1993.

[3] D.F. Brown, H. Moura, and D.A. Watt. Actress: an action semantics directed compiler generator. In *Proceedings of the Workshop on Compiler Construction*, Paderborn, Germany, 1992.

[4] W.F. Clocksin and C.S. Mellish. *Porgramming in Prolog.* Spring-Verlaug, 4th edition, 1994.

[5] K.G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, Vol.19, No. 4:213–233, 1993.

[6] K.D. Lee. *Action Semantics-based Compiler Generation.* PhD thesis, Department of Computer Science, University of Iowa, 1999.

[7] K.D. Lee. A formally verified register allocation framework. In *Science of Computer Programming - Proceeding of LDTA 2003.* Elsevier Science, 2003.

[8] P.D. Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26.* Cambridge University Press, 1992.

[9] P. Ørbæk. Oasis: An optimizing action-based compiler generator. In *Proceedings of the International Conference on Compiler Construction, Volume 786*, Edinburgh, Scotland, 1994. LNCS.

[10] J. Palsberg. A provably correct compiler generator. In *Proceedings of the 4th European Symposium on Programming (ESOP92).* LNCS, 1992.

[11] T. Pittman and J. Peters. *The Art of Compiler Design.* Prentice Hall, Englewood Cliffs, NJ 07632, 1992.

[12] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison Wesley, 2nd edition, 1996.

[13] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Adisson Wesley Publishing Company, Inc., New York, NY, 1995.

[14] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, ml97 edition, 1998.

[15] D. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1991.

```
structure calc =
struct
    open calcAS;

    structure calcLrVals =
    calcLrValsFun(structure Token = LrParser.Token);
    structure calcLex =
    calcLexFun(structure Tokens = calcLrVals.Tokens);
    structure calcParser =
    Join(structure Lex= calcLex
         structure LrParser = LrParser
         structure ParserData = calcLrVals.ParserData);

    val calcparse = fn filename =>
    let val instrm = TextIO.openIn filename
        val lexer = calcParser.makeLexer(fn i => TextIO.inputLine instrm)
        val _ = calcLex.UserDeclarations.pos := ${_1}$
        val error = fn (e,i:int,_) => TextIO.output(TextIO.stdOut," line " ^
                                        (Int.toString i) ^ ", Error: " ^ e ^ "\n")
    in calcParser.parse(30,lexer,error,()) before TextIO.closeIn instrm
    end

    fun evaluate(add'(e1,e2),min) =
        let val (r1,mout1)= evaluate(e1,min)
            val (r2,mout) = evaluate(e2,mout1)
        in
          (r1+r2,mout)
        end

      | evaluate(sub'(e1,e2),min) =
        let val (r1,mout1)= evaluate(e1,min)
            val (r2,mout) = evaluate(e2,mout1)
        in
          (r1-r2,mout)
        end

    fun interpret filename  =
        let val (prog, _) = calcparse filename
        in
            let val (result,mout) = evaluate(prog,0)
            in
              TextIO.output(TextIO.stdOut,"= "^Int.toString(result)^"\n")
            end
        end

    fun run(a,b::c) = (interpret b;
               OS.Process.success)
      | run(a,b) = (TextIO.print("usage: sml @SMLload=ram filename\n");
            OS.Process.success);
end
```

Figure 12: Calculator Functions to Call the Interpreter