

Evaluating Inherited Attributes Using Haskell and Lazy Evaluation

William B. Moss
Haverford College
Department of Computer Science

Advised by: Dr. David G. Wonnacott

May 6, 2005

Abstract

Parser generators designed for imperative programming languages (C++, Java, etc), such as YACC and CUPS, inevitably run into difficulty handling inherited attributes. In the past, this difficulty has been handled in one of two ways: 1) the problem is ignored, and inherited attributes must be manually calculated later using extra passes over the parse tree; and 2) for the L-attributed class of grammars, the parser generator provides some sort of "hack," allowing the user to access previous elements of the stack with no safeguards preventing a user from accessing incorrect data. Many functional programming languages, however, provide an easy solution to this problem. Using lazy evaluation, a parser generator written for a function programming language can sidestep the entire problem, relying on the language system to evaluate attributes (regardless of whether they are inherited or synthesized) as it needs them. This paper discusses a method for evaluating inherited and synthesized attributes in Haskell, specifically using Happy, a parser generator for Haskell. This paper will present specific examples of parsing inherited attributes in both functional and imperative languages, followed by the presentation of an algorithm for computing any inherited attribute in Happy.

Acknowledgments

I would like to thank Dave Wonnacott for his guidance in the creation of this document, as well as the rest of the Haverford CS department for their continued dedication to my education.

Contents

1	Introduction to Imperative and Functional Programming	4
1.1	Imperative Programming	4
1.2	Functional Programming	5
1.2.1	Non-Strict Evaluation	5
1.2.2	Lazy Evaluation	7
1.2.3	Unevaluated Functions and Lambda Abstractions	7
2	Introduction to parsing and parser generators	9
2.1	Introduction	9
2.2	Introduction to Parsing	10
2.3	LR Parsing	11
2.4	Tree Representations	12
2.5	Synthesized attributes	12
2.5.1	Multiple Attributes	13
2.6	Trees with Attributes	14
2.7	Inherited attributes	14
3	Parsing and Evaluating Inherited Attributes	16
3.1	Grammar 2.4 Revisited	16
3.1.1	Global Symbol Table	16
3.1.2	L-attributed Grammars	17
3.1.3	Traversing Back on the Stack	18
3.2	Adding More Variable Declarations	19
3.3	Forward Reference Variable Declarations in YACC	20
3.4	Evaluating Inherited Attributes in Haskell and Happy	22
3.4.1	Using let to simplify Vars	24
3.5	Evaluating Inherited Attributes in Miranda/SNACC	25

4	Generalizing an Algorithm for Inherited Attributes in Happy	27
4.1	Using Multiple Named Attributes	27
4.2	Using Lets to Simplify Multiple Attributes	28
4.3	Using Unevaluated Functions for Inherited Attributes	29
4.4	Generating HERA machine language	31
5	Conclusion	33

Chapter 1

Introduction to Imperative and Functional Programming

1.1 Imperative Programming

Imperative programming is, perhaps, the most intuitive method of programming. In imperative programming, statements are executed one-by-one in the order specified by the program. The following example illustrates a simple function in an imperative language:

```
[1] int PlusOne (int x)
[2]   x = x + 1
[3]   return x
```

Figure 1.1: A Simple Function in C++

In this example, a function (`PlusOne`) takes one integer parameter and returns an integer as well. In line 2, the program increments the parameter (`x`) and then in line 3 returns the incremented value. Although this example is quite trivial, it exemplifies a general underpinning of imperative programming; the statements must be executed in order so that the correct value is achieved. Consider what would happen if the program executed lines $1 \rightarrow 3 \rightarrow 2$ instead of lines $1 \rightarrow 2 \rightarrow 3$. The function would begin normally; however, the value would be returned immediately following that. Although the value of `x` would be incremented in the third step, it would already have been returned and therefore the increment would have no effect.

It is clear that the compiler must execute the lines in the order specified by the programmer or the results may be unpredictable.

1.2 Functional Programming

Function programming provides a different paradigm for thinking about and writing programs. In functional programming, programs are specified by a series of functions (much like in mathematics). Unlike imperative languages (which are almost uniformly compiled), functional languages are sometimes compiled and sometimes interpreted. For simplicity in this paper, I will just refer to the language system or language, since regardless of the method of execution; the code will return the same value. Grammar 1.2, presents `PlusOne` in a functional language:

```
PlusOne x = x + 1
```

Figure 1.2: A Simple Function in Haskell

Although the syntax is slightly different, both functions do exactly the same thing. It is important to note, however, that there is no longer a well defined order of execution—we cannot visualize the program stepping from one line to the next as we can with an imperative language. Giving this control to the language does, however, allow functional languages to provide a number of other useful features that are not available in imperative languages. The most important of which are non-strict evaluation, lazy evaluation and returning unevaluated functions. These are discussed in the next three sections.

1.2.1 Non-Strict Evaluation

Due to the fact that a functional programming language defines only functions, and therefore only results, the programmer has no control over the order in which commands are executed. This is very different from imperative programming, where the programmer explicitly defines the control flow. Although this may seem like a shortcoming of functional languages, giving the language the control over the order in which operations are executed also gives the language control over whether an operation is executed at all. Grammar 1.3 presents a simple example in an imperative language.

```
[1]int PlusOne (int x, int y)
[2]  x = x + 1
[3]  return x

[4]int Infinite (int x)
[5]  while (true)
[6]    x = x + 5
[7]  loop
[8]  return x
```

Figure 1.3: A Non-Terminating Function in C++

This is the same `PlusOne` from before, except we have added a second parameter that is not used by the function. We have also added a second function that will go into an infinite loop, never returning a value. Now that we have these functions, what happens if we try to calculate `PlusOne(2, Infinite(3))` in an imperative language? Since the control flow is dictated by the programmer, the compiler must evaluate all the arguments to the function before continuing to the next line. The compiler has no trouble evaluating the first argument to `PlusOne`, however, upon trying to evaluate the second argument (`Infinite(3)`), the program enters an infinite loop. The answer, therefore, is never found and the program never terminates.

What happens, however, if we try the same thing in a non-strict functional language?

```
PlusOne x y = x + 1

Infinite x = Infinite (x + 5)
```

Figure 1.4: A Non-Terminating Function in Haskell

The functional definitions appear quite similar to the imperative one; however, since the language chooses the order in which elements are evaluated, it can get around the problem faced by the imperative language. It goes into `PlusOne` and realizes it needs the first parameter (`x`) so it evaluates it (again, quite simply). It then adds one and returns the value. Since the language was dictating the order in which elements were evaluated (and not the programmer) the language was able to avoid evaluating the second parameter to the function and thereby avoid getting stuck in an infinite loop. Although this example is contrived, and the imperative program could clearly be fixed by any decent programmer, this concept will be exceptionally important later.

1.2.2 Lazy Evaluation

Lazy evaluation is the deliberate delaying of the evaluation of functions and data structures until they are needed. This provides the programmer with an excellent extension to non-strict evaluation—the ability to use part of an infinite data structure. Grammar 1.5 presents an example of an infinite data structure (in this example we introduce the `:` operator, which concatenates an element onto the head of a list a list).

```
ints x = x : ints (x + 1)
```

Figure 1.5: An Infinite List

One can see that the function `ints` creates an infinite list starting with the initial parameter and counting up. Consider another function called `take n l`, which has two arguments: `n` is the number of elements to take from the beginning of list `l`. For example, `take 2 [1, 2, 3, 4, 5] ⇒ [1, 2]`. Now consider `take 3 (ints 1)`. We know that if `take` does not use the second parameter everything will work (the language will just not evaluate `ints 1`). This, however, is not the case—`take` uses `ints 1`, and in this case must remove the first three elements from it. Nevertheless, this should be possible; the first three elements of `ints 1` are clearly defined and not infinite. In a lazy functional programming language, infinite data structures can be used, so long as they are used in a way that generates a finite result. This allows us to not only evaluate `PlusOne 3 (ints 1) ⇒ 4`, but also to evaluate `take 3, (ints 1) ⇒ [1, 2, 3]`.

1.2.3 Unevaluated Functions and Lambda Abstractions

In most functional programming languages, including Haskell, it is possible to return unevaluated functions, just as you would any other data type. Lambda abstractions provide an excellent method for simplifying the syntax for returning unevaluated functions. If we wish to make `add` the return value of a function, we can construct the function `add` and then return it. If we are only looking to use it once, however, it seems unnecessary to construct the function separately before returning it. The lambda abstraction equivalent of `add` is simply `\x y -> x + y`. This creates a function that takes two

parameters and adds them (just like `add`). Although we cannot refer to this function by name, if we want to use it to evaluate $3 + 5$, we could simply write `\x y -> x + y) 3 5` and if we wanted to return it from a function, we could simply have the function return `\x y -> x + y) 3 5`.

Chapter 2

Introduction to parsing and parser generators

2.1 Introduction

When a program is written in a higher level language like C++, Java or Haskell it must be compiled into binary that can then be directly executed by the processor. Higher level languages offer a number of advantages over programming directly in assembly, the most important of which are the fact that the programmer can focus on the higher level algorithms (and not the gritty details of assembly) and that higher level languages are generally processor independent. Programming in a higher level language, however, requires that someone write a compiler for the language. Converting a program to assembly is a process easier thought of in terms of pattern matching and the grammatical structure of the languages. Since new programming languages are often developed and many of the steps in converting any programming language to assembly are the same, the necessary algorithms have been studied in detail. Most compilers for new languages are written in other higher level languages and many compiler writing tools have been written for most modern programming languages.

The first step in compiling any program is called tokenizing it. This involves matching various character patterns and replacing them with a token (i.e. "if" would be replaced with the IF token or a consecutive set of digits would be replaced with an integer token). This tokenization process is handled by a program called a lexical scanner (or lexer). The user defines

a series of patterns and what token they correspond to, and the lexer will generate code in the language it is designed for (C++ programmers use Lex and Alex exists for Haskell).

The list of tokens is then passed off to another piece of code that is generated by a different compiler tool, called a parser generator or a compiler compiler. A parser generator is a program that takes a grammatical specification for a program and generates a series of functions capable of parsing that grammar. This paper will focus of the difference between parser generators in various languages, specifically focusing on the differences between parser generators for imperative versus functional programming languages. Parser generators exist for almost every language, but for the purposes of this paper only three are considered, YACC, Happy and SNACC, which produce code for C++, Haskell and Miranda, respectively.

2.2 Introduction to Parsing

To better understand how parsing works, the code below illustrates a very simple grammar that would be used for parsing expressions involving adding and subtracting of integers

```
Exp  : Exp '+' Exp
      | Exp '-' Exp
      | '(' Exp ')
      | int
```

Grammar 2.1: Simple Arithmetic Grammar

This grammar contains five terminal symbols, one non-terminal and four productions. Terminal symbols are tokens received from the lexer (+, -, (,) and int) whereas the non-terminals are symbols used by the grammar to allow for the rules to be applied multiple times (**Exp**). A production is one of the statements in the right column (i.e. **Exp '+' Exp** or **int**). This grammar will match any correct arithmetic expression using only addition, subtraction and parentheses and will fail to match an improperly formed arithmetic expression (i.e. it will match $(2+3)-6$, but will not match $3++4$). Grammars function by replacing symbols in the input with productions and then replacing the productions with non-terminal symbols. For example, in the expression $2+3$, 2 and 3 would each be lexically analyzed to **int**, leaving **int '+' int**. The two **ints** would be replaced by **Exps** and then

the `Exp '+' Exp` would be replaced with `Exp`, resulting in a successful parse (since we were able to turn the input into a single `Exp`). The next section goes into a more formal explanation of how a parser generator actually parses a specific input.

2.3 LR Parsing

LR parsing is the standard method for parsing input using a grammar. LR stands for left-to-right parse, rightmost-derivation in reverse. This means that the input is read from left to right (as one might expect). The reverse means that it is taking productions on the right side and replacing them with the correct non-terminals, as opposed to replacing non-terminals with the correct production and the rightmost means the rightmost production that can be replaced is. In the implementation of parsing, a stack is used to store the terminals and non-terminals that have been read from the input but have not yet been matched and replaced. LR parsing also introduces two new terms, shift and reduce, to describe the motion of information onto and off of the stack. Shifting is reading a character from the input and placing it on the stack. Reducing is the application of a grammar rule by matching elements on the stack to a production; popping those elements off of the stack and pushing the correct non-terminal back onto the stack. Figure 2.1 shows a sample parse of the input $(2 + 3) - 6$ (for full details on LR parsing, consult the lectures from CMSC 350 or [1]).

Stack	Input	Action
	$(2 + 3) - 6 \$$	shift
($2 + 3) - 6 \$$	shift
(int	$+ 3) - 6 \$$	reduce <code>Exp -> int</code>
(Exp	$+ 3) - 6 \$$	shift
(Exp +	$3) - 6 \$$	shift
(Exp + int	$) - 6 \$$	reduce <code>Exp -> int</code>
(Exp + Exp	$) - 6 \$$	reduce <code>Exp -> Exp '+' Exp</code>
(Exp	$) - 6 \$$	shift
(Exp)	$- 6 \$$	reduce <code>Exp -> '(' Exp ')'</code>
Exp	$- 6 \$$	shift
Exp -	$6 \$$	shift
Exp - int	$6 \$$	reduce <code>Exp -> int</code>
Exp - Exp	$\$$	reduce <code>Exp -> Exp '-' Exp</code>
Exp	$\$$	accept

Figure 2.1: LR Parser Stack

2.4 Tree Representations

Trees provide an excellent way of representing the parsing of an input by a grammar. As the previous section demonstrates, patterns are matched and then during a reduce operation, part of the input is replaced by a simpler non-terminal. In the case of a tree representation, the reduce action is equivalent to moving up the tree. Figure 2.2 shows the tree for the parse of $(2 + 3) - 6$.

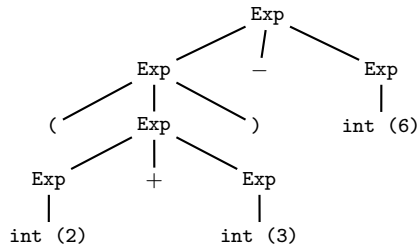


Figure 2.2: Parse Tree for $(2 + 3) - 6$

Although it is useful to represent the entire parse in a tree, much of the information represented there is repetitive and not necessary. In Figure 2.3 much of the superfluous information has been stripped away, leaving a smaller more efficient tree representation. This simpler representation is referred to as an abstract syntax tree.

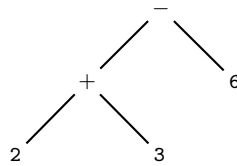


Figure 2.3: Abstract Syntax Tree for $(2 + 3) - 6$

2.5 Synthesized attributes

Although it is useful to use a grammar to match input and determine whether or not the input is valid, parsing becomes vastly more useful when we can attach a piece of code to each production. In Grammar 2.2, the value of the input is calculated on the fly using simple arithmetic operations. The

convention (originally from YACC and adopted by Happy) that will be used henceforth in this paper is to refer to the value of each symbol in a production as a \$ followed by the number that symbol is in the production. The count is one based, so in the first line below, the \$1 refers to the first `Exp` and the \$3 refers to the second `Exp` (\$2 would refer to the `+`, which has no value).

```
Exp  : Exp '+' Exp    { $1 + $3 }
      | Exp '-' Exp   { $1 - $3 }
      | '(' Exp ')'   { $2 }
      | int           { $1 }
```

Grammar 2.2: Defining Attributes

By adding a method for computing values associated with each production parsers can not only match an input program to a given grammar (and return true or false depending on whether the program is syntactically correct) but can now associate pieces of code with the various productions, allowing for the return of any number of values beyond just true and false.

2.5.1 Multiple Attributes

Some parser generators provide the user with the option of having multiple synthesized attributes for each expression. YACC distinguishes between the various attributes by naming them. Although this is very convenient syntactically, it provides not more power than allowing for one record type attribute. In this example we also introduce \$\$, which is the code used to describe the non-terminal on the left side of the production. Grammar 2.3 shows a modified version of Grammar 2.2 to include a second attribute that counts the number of expressions.

```
Exp  : Exp '+' Exp    { $$ .val = $1 .val + $3 .val, $$ .num = $1 .num + $3 .num + 1 }
      | Exp '-' Exp   { $$ .val = $1 .val - $3 .val, $$ .num = $1 .num + $3 .num + 1 }
      | '(' Exp ')'   { $$ .val = $2 .val, $$ .num = $2 .num + 1 }
      | int           { $$ .val = $1, $$ .num = 1 }
```

Grammar 2.3: Defining Multiple Attributes

Here we can clearly see that `val` will contain the value of the expression (just as in the previous example), but now we have added `num`, which counts the number of expressions that comprised the input. The concept is, of course, almost entirely identical to the previous situation, we have just

added slightly more functionality by adding multiple attributes that can be associated with each production in the grammar.

2.6 Trees with Attributes

Now that attributes can be associated with the productions in a grammar, consider how this effects the tree representation. Since each production in the grammar represents one node, a value can be associated with each node in the tree. For example, Figure 2.4 shows the tree representation of the parse of $(2 + 3) - 6$ with Grammar 2.2.

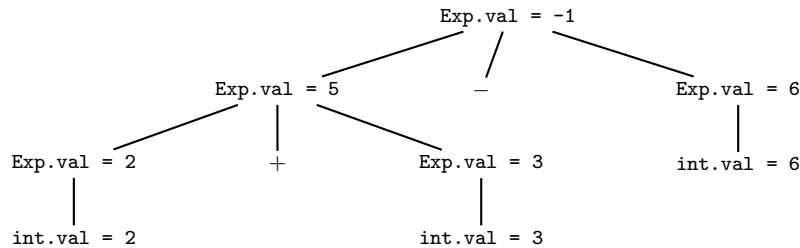


Figure 2.4: Associating Values with a Parse Tree

This can also be applied to the nodes of the abstract syntax tree structure, which is shown in Figure 2.5.

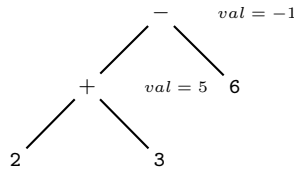


Figure 2.5: Simplified Parse Tree with Associated Values

Like the original tree representation, this provides an excellent way of visualizing the parsing of an input by a grammar.

2.7 Inherited attributes

The addition of attributes for each production greatly increases the power of parsing and parser generators. The example in the previous section, how-

ever, deals only with one type of attribute—attributes that move from the right to the left side of a production. These attributes, called synthesized attributes, only move data in one direction. In Grammar 2.4, however, we see that synthesized attributes will not be able to compute the necessary values. Below we have added a `let` expression, allowing the user to initialize a variable with a certain value and then use it later in the program.

```

Exp  : let var '=' Exp 'in' Exp  { ??? }
     | Exp '+' Exp              { $1 + $3 }
     | Exp '-' Exp              { $1 - $3 }
     | '(' Exp ')'              { $2 }
     | int                      { $1 }
     | var                      { ??? }

```

Grammar 2.4: Adding a let expression

The `let` allows a user to define a variable and initialize it with a given value, and the `var` allows the user to use that variable. It is unclear; however, what kind of code would be attached to these productions to complete the necessary steps. Somehow, the information about the variables defined in the `let` needs to travel down the second `Exp` ($\$6$). If this information does not pass down into the second `Exp` there is no way a `var` in that `Exp` could know what variables have been defined and which have not.

To solve this problem, we can imagine some system in which a list of the available variables is assigned as an attribute to the second `Exp` and that is continuously passed down to whatever expression that `Exp` represents. Then a `var` expression could do a lookup in that list and find the value of that variable (or that it had not been properly defined). This system would require a new type of attribute, one that can be passed from left to right (or down the tree). Attributes that are defined in terms of other attributes to the left (above or on the same level in the tree) are called inherited attributes.

Inherited attributes, unlike synthesized attributes, are not simple to implement in a parser generator. Remembering back to the example of an LR parser, the parser popped elements off the stack and replaced them in a reduce operation. The calculation of a synthesized attribute can easily be done by applying the code for the production as the reduce is executed and then just associating the calculated value with the new non-terminal that is placed back on the stack. The remainder of this paper will explore ways in which parser generators designed for imperative versus functional programming languages tackle the problem of calculating inherited attributes.

Chapter 3

Parsing and Evaluating Inherited Attributes

Grammar 2.4 in Section 2.7 presented the problem of evaluating inherited attributes, the following sections will present solutions to that grammar and a slightly more complicated grammar in both YACC and Happy. Since Happy and Haskell provide more extensible solutions for evaluating inherited attributes, the first sections will deal with YACC and C++. As the complexity of the inherited attributes is increased, the solutions in YACC will become more and more complicated and Happy will be introduced as an easier and more efficient solution to the problem.

3.1 Grammar 2.4 Revisited

3.1.1 Global Symbol Table

As discussed in Section 2.7, the code that is associated with the `let` and `var` productions must add variable definitions to a symbol table and lookup variables in the symbol table, respectively. The first and simplest solution to this problem is not to pass that information around in the parser, but rather to create a symbol table that is stored in a global variable. Due to the fact that the symbol table is global, variables must be removed when they move out of scope as well as being added when they are in scope. To accomplish this, three functions are added: `insert`, `remove` and `lookup`. These functions insert a value into the global symbol table, remove a value from the symbol

table and look up a value, respectively. Grammar 3.1 presents Grammar 2.4 with the addition of these functions.

```

Exp  : let var '=' Exp 'in' { insert ($2, $4) } Exp      { remove ($2, $4) }
      | Exp '+' Exp                                     { $1 + $3 }
      | Exp '-' Exp                                     { $1 - $3 }
      | '(' Exp ')'                                     { $2 }
      | int                                             { $1 }
      | var                                             { lookup ($1) }

```

Grammar 3.1: Global Symbol Table

To represent the fact that the variable must be added to the symbol table before the second `Exp` is parsed, the code for `insert` is directly after the `'in'`. The `remove` is then placed after the `Exp`, once the variable is no longer in scope. Grammar 3.1 provides a solution to our problem, however, the global symbol table solution is far from an elegant solution. Not only is it a pain to be constantly adding and removing entries from a global variable, but global variables always run the risk of being modified in unexpected ways by code written by other programmers.

3.1.2 L-attributed Grammars

Another solution utilizes evaluation techniques for a certain subclass of grammars called L-attributed grammars. A grammar is L-attributed if, for every production $A \rightarrow X_1X_2\dots X_n$, the inherited attributes of X_j only depend on the attributes (both synthesized and inherited) of $X_1X_2\dots X_{j-1}$ (i.e. the non-terminals to the left) and A . Once a grammar is classified as L-attributed, a slightly different type of parsing can be used to evaluate the inherited attributes.

In an L-attributed grammar, the code is not only associated with full productions, but also within individual symbols within a production. Code within a production is executed after the symbol on its left is parsed. Grammar 3.2 presents Grammar 2.4 with productions given in the format an L-attributed parser would be able to evaluate.

Although Grammar 3.2 seems to have productions everywhere, on closer inspection the productions at the end of the line are the synthesized attributes and the productions in the middle are the inherited attributes. The productions involving inherited attributes each assign the value of the inherited attribute immediately before that attribute. The LL parser guarantees

```

Exp  : let var '=' Exp 'in' { $6.st = insert ($$.st, $2, $4) } Exp      { $$ .val = $6.val }
    | { $1.st = $$ .st } Exp '+' { $3.st = $$ .st } Exp              { $$ .val = $1.val + $3.val }
    | { $1.st = $$ .st } Exp '-' { $3.st = $$ .st } Exp              { $$ .val = $1.val - $3.val }
    | '(' { $2.st = $$ .st } Exp ')'                                   { $$ .val = $2.val }
    | int                                                            { $$ .val = $1.val }
    | var                                                            { $$ .val = lookup ($1.st, $1) }

```

Grammar 3.2: L-attributed Grammar

that any non-terminal in a production is completely parsed before moving on to the next symbol. This ensures that when the code that assigns the inherited attribute to a symbol is executed (which happens immediately before the symbol is parsed), the attribute's value can be placed on the top stack without fear of it being replaced or modified. Since all the non-terminals on to the left have already been reduced, the next change to the stack must be the addition of the non-terminal whose inherited attribute was just set.

3.1.3 Traversing Back on the Stack

YACC, unfortunately, will not parse an L-attributed grammar written in the form of Grammar 3.2. It is possible, however, to use YACC to parse any L-attributed grammar using a feature that allows the user to access elements on the stack. As discussed in section 1.3, an LR parser uses a stack to store the current state of the parse (the various terminals and non-terminals that still need to be matched). YACC allows users to access back in the stack; however, it provides no safeguard as to what a user might find there. To do this, the user types \$-1, \$-2, etc. signifying the element one, two, etc. below the current element on the stack. I will not go into a detailed discussion of how one would convert an L-attributed grammar into something that could be evaluated by accessing previous elements on the stack, but I would like to note that this method is undesirable for a number of reasons.

As discussed before, the goal of higher level languages is to move the programmer away from the specific implementations and idiosyncrasies of their specific system and towards a general method for defining and reasoning about algorithms. This, clearly, violates that tenant of higher level languages, as the programmer must worry about the order of elements on the stack. The most egregious problem, however, lies in the fact YACC provides no checking of any sort. This means that not only is the programmer required to be 100% sure of what they are coding (for fear of causing a random crash), but

if someone else were to come along later and modify their grammar without full knowledge of where they has used $\$-1$ and $\$-2$ they could create a situation in which $\$-1$ and $\$-2$ were no longer defined.

3.2 Adding More Variable Declarations

Grammar 2.4 presents an interesting problem and provides a good introduction to inherited attributes, however, what if the user wishes to declare more than one variable with a `let` statement? Grammar 3.3 presents Grammar 2.4 with the addition of three variable declarations instead of just one.

```
Exp  : let var '=' Exp ',' var '=' Exp ',' var '=' Exp in Exp
      | Exp '+' Exp
      | Exp '-' Exp
      | '(' Exp ')
      | int
      | var
```

Grammar 3.3: Three Variable Declarations

Although this grammar is fully functional, it will be useful to simplify the grammar slightly to ease in the writing of the associated code. Grammar 3.4 shows Grammar 3.3 with the new non-terminal added. This way, `Vars` can return information about the three variables that are declared and `let` can apply those new variables to the `Exp`.

```
Exp  : let Vars in Exp
      | Exp '+' Exp
      | Exp '-' Exp
      | '(' Exp ')
      | int
      | var

Vars : var '=' Exp ',' var '=' Exp ',' var '=' Exp
```

Grammar 3.4: Grammar 3.3 Simplified

First consider Grammar 3.4 where it only allows backward referenced variable declarations. This means that `let a = 3, b = 4, c = a + b + 1 in f(a, b, c)` will be allowed, however, `let a = b + c - 3, b = 4, c = 1 in f(a, b, c)` will not. This grammar, although slightly more complicated, can be evaluated using the same techniques as Grammar 3.1. Since

the variables can only backward references, we still know every variable declared will be in scope for all other variables to the right of it in the input, so the global variable method would work. It could also be evaluated using L-attributed methods since the inherited attributes for the second and third Exps in Varonly depend on elements to their left.

3.3 Forward Reference Variable Declarations in YACC

Now consider the final level of complexity—forward reference variable definitions in the `let`. Now, not only will `let a = 3, b = 4, c = a + b + 1 in f(a, b, c)` be allowed, but, `let a = b + c - 3, b = 4, c = 1 in f(a, b, c)` will be as well. Unfortunately, YACC no longer provide the user with a simple way to calculate the values of the `let` and `var` expressions completely within the grammar. To evaluate these attributes in YACC, a tree of the input is created. This tree, although a very useful representation of the input, will require a decent amount of post processing to actually produce a result for any given input. Grammar 3.5 shows the associated code in YACC to construct a tree from the input.

```
Exp  : let Vars in Exp      { $$ast = new letExp($2.vars, $4.ast) }
    | Exp '+' Exp         { $$ast = new opPlus($1.ast, $2.ast) }
    | Exp '-' Exp         { $$ast = new opMinus($1.ast, $2.ast) }
    | '(' Exp ')'         { $$ast = $2.ast }
    | int                 { $$ast = new intExp($1) }
    | var                 { $$ast = new varExp($1) }

Vars : var '=' Exp ',' var '=' Exp ',' var '=' Exp
      { $$vars = new varDec($1, $3, $5, $7, $9, $11) }
```

Grammar 3.5: Recursive Grammar in YACC

Since this grammar is constructing a tree from the input, a convenient way to understand it will be to run it on a sample input. Figure 3.1 displays the parse of: `let x = z - 1, y = 4, z = y + 3 in (x + 3) - y`.

This tree would then be passed to another set of functions that would recursively traverse the tree and eventually return the result of the input. Since the focus of this paper is not parsing in YACC, I will not go into specific detail about how one would write the functions, however, one can imagine two functions, one that passed information about declared variables up the

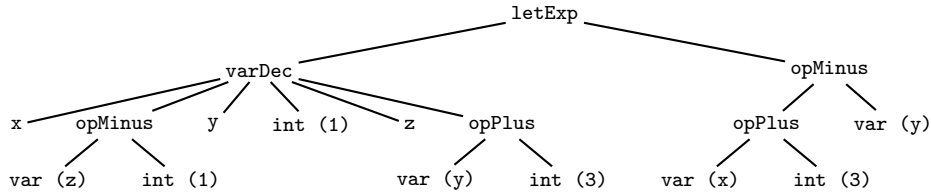


Figure 3.1: Parse of Sample Input in YACC

tree and another that passes information about available variables down the tree. The logical first step for these functions is to figure out what variables are declared and what their values are. Due to the fact that the variable declarations can be mutually recursive, the functions would need to traverse the declarations section five times: once up (with a value for only one of the three variables), then down, then up again (this time adding one variable that's value was based on the information that was just passed down), then down again, then finally back up with all the information. This fully constructed variable list, called a symbol table, could then be passed down the expression side of the tree. Finally, the value of the input could be calculated by traversing up the expression side of the tree doing the arithmetic and looking up variables as necessary. Although this grammar is rather simple, one `let` expression results in seven traversals of the tree.

To generalize this to an more arbitrary grammar, for a `let` with k variable declarations, $k - 1$ traversals would be required to evaluate all the declared variable. So, an input with j `let` expression each with k variable declarations, would require $j(k - 1) + 2$ traversals of the tree (the $+2$ is the result of the final traversals to pass the declared variables down the tree and to bring the value of the synthesized attribute back up the tree).

Although tree traversals are a useful measure of complexity, it will be more useful to break it down to into the number of basic operations that are required, in this case, following a pointer. In the worst case, if we have an input that has a total of n variable declarations, it will create an abstract syntax tree of order n . Again, in the worst case, each variable declaration will depend on other variable declarations such that n traversals of the tree will be necessary to evaluate all the variable values. So, n traversal of a tree of order n , results in a complexity of $O(n^2)$.

To generalize even farther, we can also say that the upper bound for post-processing an abstract syntax tree is also $O(n^2)$. By similar logic, if there are

n pieces of data in the input that need to be analyzed, it should be possible to represent that data in an abstract syntax tree of order n . If there are m attributes at every node, it should be possible to define one of those m pieces in every pass of the tree. Since attributes need only be defined once, in the worst case, we need to make mn passes over a tree of order n , again giving a complexity of $O(mn^2)$, which is equivalent to $O(n^2)$ since m is a constant for any given abstract syntax tree.

3.4 Evaluating Inherited Attributes in Haskell and Happy

The solution proposed in Section 3.3, although effective, clearly takes a good deal of extra code for certain types of inherited attributes. It would be nice to have a general solution that would allow for the evaluation of arbitrary inherited attributes without any post-processing. To do this, a lazy functional language will have to be used—we will use Happy and Haskell. Grammar 3.6 presents Grammar 2.4 in Happy. We start with Grammar 2.4 for simplicity, however, unlike in YACC, it will be very easy to extend the solution to evaluate Grammar 3.4.

```
Exp  : let var '=' Exp in Exp { \p -> $6 ($2, $4):p }
    | Exp '+' Exp           { \p -> $1 p + $3 p }
    | Exp '-' Exp           { \p -> $1 p - $3 p }
    | '(' Exp ')'           { $2 }
    | int                   { \p -> $1 }
    | var                    { \p -> case lookup $1 p of
                                Nothing -> error "no var"
                                Just i  -> i }
```

Grammar 3.6: Grammar 2.4 in Happy

Each of the productions in `Exp` returns a function from an environment, `p`, to an integer value. Taking the simplest production in the grammar, `int` ignores the environment and just returns the value of the integer it parses. This makes sense, as an integer value is fixed and should not be changed by the surrounding code (i.e. the environment). Next consider plus (minus, of course, is identical just with a different arithmetic operation). It takes an environment and passes it as a parameter to the two `Exps` that make up the plus expression. Again, this makes sense, since each `Exp` returns a function that takes an environment and returns a value, if we pass the environment to

each of the `Exps`, the unevaluated function will evaluate to an integer value. We can then just add those two integer values together to get the desired result.

Now consider the construction and use of the environment, `p`, and how that enables the evaluation of `let` and `var`. The environment, `p`, is really just a simple symbol table, a list of string-integer pairs. The string-integer pairs represent variables and their associated value. Looking at `var`, we see it just calls the function `lookup`, a function that takes two arguments, a symbol to search for (`$1`) and a list to search in (`p`). So `var` just does a lookup in the symbol table, and returns a function that either returns the value of that variable if it exists, or returns `error "no var"` if it does not.

The construction of the environment takes place in the `let` expression. As was stated before, the environment, `p`, is just a list of string-integer pairs, so a `let` expression must modify the environment to include the newly defined variables and then pass that new environment on to its `Exp`. This is, of course, exactly what the `let` does. It constructs the new string-integer pair, (`$2`, `$4`), and then concatenates it onto the head of the symbol table `p`. This new list is then passed as the parameter to the second `Exp`, `$6`.

Now that Grammar 3.6 is understood, consider Grammar 3.7.

```
Exp  : let Vars in Exp      { \p -> $4 ($2 p) }
    | Exp '+' Exp         { \p -> $1 p + $3 p }
    | Exp '-' Exp         { \p -> $1 p - $3 p }
    | '(' Exp ')'         { $2 }
    | int                  { \p -> $1 }
    | var                   { \p -> case lookup $1 p of
                                Nothing -> error "no var"
                                Just i   -> i }

Vars : var '=' Exp ',' var '=' Exp ',' var '=' Exp
    { \p -> (($1, $3 (($5, $7 (($9, $11 p):p))):(($9, $11 (($5, $7 p):p)):p)):
      (($5, $7 (($1, $3 (($9, $11 p):p))):(($9, $11 (($1, $3 p):p)):p)):
      ($9, $11 (($1, $3 (($5, $7 p):p))):(($5, $7 (($1, $3 p):p)):p)):p) }
```

Grammar 3.7: Grammar 3.5 in Happy

From Grammar 3.6 to Grammar 3.7, the only change is the addition of the non-terminal `Vars`. Upon closer inspection, `Vars` just returns a function that takes an environment and returns a new environment with all the new variable declarations included, the entire production for the `let`, then just passes the old environment to `Vars`, which returns the new environment which is then passed to the second `Exp` as before. Due to the three mutually

recursive variable declarations, the code for `Vars` gets a little messy. The complexity arises from the fact that the environment variable that is passed to each of the variables in `Vars` cannot just be in the incoming environment, but must also include the information from the other two variable declarations. This creates the very complicated statement for `Vars`. I will not go into a lot of detail about the intricacies of this function as it is simplified in the next section.

3.4.1 Using `let` to simplify `Vars`

Consider Grammar 3.8, in which `Vars` has been modified (all other productions remain the same).

```

Exp  : Exp '+' Exp      { \p -> $1 p + $3 p }
      | Exp '-' Exp     { \p -> $1 p - $3 p }
      | '(' Exp ')'     { $2 }
      | int              { \p -> $1 }
      | let Vars in Exp  { \p -> $4 ($2 p) }
      | var              { \p -> case lookup $1 p of
                              Nothing -> error "no var"
                              Just i   -> i }

Vars : var '=' Exp ',' var '=' Exp ',' var '=' Exp { doVars $1 $3 $5 $7 $9 $11 }

doVars v1 e1 v2 e2 v3 e3 p = let all    = first:(second:(third:p))
                              first  = (v1, e1 all)
                              second = (v2, e2 all)
                              third  = (v3, e3 all) in
                              all

```

Grammar 3.8: Simplifying Grammar 3.7 Using `Let`

In Grammar 3.8, nothing has changed except the code associated with `Vars`. Due to an idiosyncrasy in the Happy/Haskell interaction, a separate function had to be defined instead of just associating it directly with the production, however, the premise is the same, `Vars` will return a function that takes an environment and returns a new environment with the new variables declared in it. The `doVars` function utilizes a convenient feature of Haskell that acts almost identically to the `let` expression we have been working to define. One can associate variables with values, and then use those variables in the body of the `let`. For this grammar, four variables are defined. `all` is an environment that contains the old environment as well as three new elements (each of which are defined later in the `let` expression).

The next three variable declarations are for the three variables that can be defined in our `let` expression. As we mentioned above, since the three variables can be mutually dependant, the environment that gets passed to the `Exp` expression must contain the variable declarations for the other two variables. Conveniently, we have defined that environment previously in the `let` expression with `all`. So, `first`, `second` and `third` each create a pair with the variable name and `all` passed to the `Exp`, returning an integer. Since the function `doVars` must return a new environment containing the variable declarations, it just returns the variable `all`.

3.5 Evaluating Inherited Attributes in Miranda/SNACC

As is hopefully clear, functional languages provide an excellent opportunity to include more expressive power in our grammar. Nevertheless, the syntax we have used so far in Happy and Haskell is rather clunky. We need to define a new function for every production then pass around different parameters every time we want to evaluate one of the non-terminals in our grammar (i.e. we must always put `$3 p`, instead of just `$3`). We have used Haskell because it is the most prevalent purely functional programming language in use today, however, another purely functional language has a parser generator that actually provide for the more direct definition of inherited attributes. SNACC (Surely Not Another Compiler Compiler) was developed by D.A. Turner for use with Miranda and uses a very similar syntax to Happy, with a few notable exceptions. The most important is the handling of inherited attributes; SNACC adds a small amount of new syntax to handle inherited attributes directly within the grammar.

To define an inherited attribute, the user places the name of the attribute next to the non-terminal on the left side of the production (in Grammar 3.9, `Exp :` becomes `Exp (env) :`, where `env` is the name of the inherited attribute). The inherited attribute may then be used like any other variable in the productions and are passed seamlessly down the parse, from the non-terminal on the left to any non-terminals on the right. To redefine the value of the inherited attribute, a user places a parenthesized expression next to the non-terminal in which the inherited attribute will be modified. For example, to change the value of `env` in the `Exp` of the `let` (since the `Exp` must be aware

of the new variables declared in the `let`), `let Vars in Exp` is changed to `let Vars in Exp(env = $2)`.

```

Exp (env): Exp '+' Exp           { $1 + $3 }
          | Exp '-' Exp          { $1 - $3 }
          | '(' Exp ')'          { $2 }
          | int                  { $1 }
          | let Vars in Exp(env = $2) { $4 }
          | var                   { case lookup $1 env of
Nothing -> error "no var"
                                Just i -> i }

Vars (env): var '=' Exp(env = doVars $1 $3 $5 $7 $9 $11 env) ','
            var '=' Exp(env = doVars $1 $3 $5 $7 $9 $11 env) ','
            var '=' Exp(env = doVars $1 $3 $5 $7 $9 $11 env)
                                { doVars $1 $3 $5 $7 $9 $11 env }

doVars v1 e1 v2 e2 v3 e3 env = let all = first:(second:(third:env))
                                first = (v1, e1 all)
                                second = (v2, e2 all)
                                third = (v3, e3 all) in
                                all

```

Grammar 3.9: Inherited Attributes in SNACC

The SNACC syntax provides a far more concise notation for dealing with inherited attributes. Although, as the next section will show, the SNACC syntax provides no extra computational power; the new notation makes another step closer to allowing the programmer to worry about algorithms and not about syntax and the underlying design of the language being used.

Chapter 4

Generalizing an Algorithm for Inherited Attributes in Happy

Haskell and Happy provide the user with the exceptionally useful feature of being able to calculate inherited attributes directly within the grammar. Inherited attributes are not a feature directly provided by Happy, but rather something that can be done using some useful features of Haskell. Due to this, evaluating inherited attributes directly within a Happy grammar can be a somewhat convoluted process. The following sections will present an algorithm for taking any inherited attribute and converting it into an unevaluated function, as well as a useful method for creating multiple named attributes within a grammar. This algorithm is implicit in SNACC, the "Happy User's Guide" and "Attribute Grammars as a Functional Programming Paradigm," however, it was never explicitly described.

4.1 Using Multiple Named Attributes

Happy provides the user with the ability to return only one synthesized piece of data from a production, however, that piece of data may be of any type. Since Haskell allows for user defined data types, a user may define any data type and use it as the type returned by the grammar. Using this principle, a data type can be defined to incorporate any number of different attributes, and then that data type can be returned.

Suppose, as in the YACC Grammar 2.3 we wish to return two synthesized attributes—one with the value of the input and one with the number of

expressions used in the input. Figure 4.1 presents the data type that would be used to create this grammar in Happy.

```
data Node = Node {val :: Int, num :: Int}
```

Figure 4.1: Node Data Type

This data type has two elements (each of type `Int`) one named `val` and one named `num`, just like Grammar 2.3.

Since each production must return the data type `Node`, each production must, actually, construct a new `Node`, assigning the value of `val` and `num` at that node. The other difference between Happy and YACC is the functional notation for returning an individual attribute from a `Node`. Instead of using `$1.num` we apply the function `num` to `$1`, returning the `num` attribute. Grammar 4.1 presents the use of this data type to reconstruct Grammar 2.3 in Happy.

```
Exp : Exp '+' Exp    { Node {val = (val $1) + (val $3), num = (num $1) + (num $3)} }
    | Exp '-' Exp    { Node {val = (val $1) - (val $3), num = (num $1) + (num $3)} }
    | '(' Exp ')'     { Node {val = val $2, num = (num $2) + 1} }
    | int             { Node {val = $1, num = 1} }
```

Grammar 4.1: Multiple Attributes Revisited

Grammar 4.1 presents an excellent example of how we can use the convenient features of Haskell to allow for multiple named inherited attributes. This grammar looks almost identical to Grammar 2.3 and, clearly, will have the same functionality.

4.2 Using Lets to Simplify Multiple Attributes

Unlike compiler compilers for iterative languages (where the order of attribute evaluation is dictated in the code), the order in which Haskell will evaluate various attributes is not defined by the programmer. It is not clear, therefore, which order the attributes at a node will be calculated in and therefore how one would use one named attribute defined at a given node in another attribute at the same node. As was discussed in Section 3.3.1, `let`

statements can be used to simplify productions in Haskell, however, they are also very useful if a user wishes to use one attribute in the calculation of another attribute at the same node. Figure 4.2 presents a production at an arbitrary node using two attributes (`attr1` and `attr2`) where `attr2` depends on the value of `attr1`. It is presented in YACC syntax first as that provides the simplest method for visualizing variable definitions.

```
Node { $$.attr1 = $1.attr1 + $2.attr2 * $2.attr1 + $3.attr1,
      $$.attr2 = $$.attr1 + $1.attr2 * $2.attr2 }
```

Figure 4.2: Interdependent Attributes in YACC

Although the user might often like to use code like this, where one attribute is defined in terms of another, Happy does not allow for this directly within the syntax of the grammar. Since a user may still wish to do this, there are two other options: 1) recalculate the value of `attr1` within `attr2` (i.e. replace `$$.attr2 = $$.attr1 + $1.attr2 * $2.attr2` with `$$.attr2 = $1.attr1 + $2.attr2 * $2.attr1 + $3.attr1 + $1.attr2 * $2.attr2`), or 2) use a `let` statement. Figure 4.3 presents Figure 4.2 in Happy, utilizing a `let` expression to allow for the use of `attr1` in the calculation of `attr2`.

```
let thisAttr1 = (attr1 $1) + (attr2 $2) * (attr1 $2) + (attr1 $3)
    thisAttr2 = thisAttr1 + (attr2 $1) * (attr2 $2)
in Node {attr1 = thisAttr1, attr2 = thisAttr2}
```

Figure 4.3: Interdependent Attributes in Happy

4.3 Using Unevaluated Functions for Inherited Attributes

As was presented in Grammars 3.7 and 3.8, unevaluated functions can be used to calculate the value of an inherited attribute. This process is, in fact, quite simple. The important difference between a standard inherited attribute and an unevaluated function is that a function must take at least one argument (in the case of an unevaluated function being used to evaluate an inherited attribute, the parameter is the inherited attribute) and return

a value (the associated synthesized attribute). This means that we must not only define the inherited attribute, but we must also define the synthesized attribute that will depend on this inherited attribute. Once the inherited attribute and the synthesized attribute have been decided upon, it is quite simple to convert them into a series of functions. Grammar 4.2 introduced a general series of productions, including one inherited attribute (`inher`) and one synthesized attribute (`synth`), for a simple grammar.

```
Exp  : Exp '+' Exp  { $$synth = f($1.synth, $3.synth, $$inher),
                    $1.inher = g($1.synth, $3.synth, $$inher),
                    $3.inher = h($1.synth, $3.synth, $$inher) }
    | int           { $$synth = i($1.val, $$inher) }
```

Grammar 4.2: Arbitrary Inherited Attributes in YACC-like Notation

As has been stated before, to use this grammar in Happy, a function must be constructed that relates the inherited attribute to the synthesized attribute. This function can then be returned by each of the productions in the grammar. For each non-terminal in a production, the inherited attribute is passed as the parameter to the non-terminal (since all non-terminals are functions from the inherited attribute to the synthesized attribute), and then the result of this calculation can be used to calculate the value of the production. For terminals, the parameter's value is either used or ignored, but either way, the function returns a value and stops the continued passing of the inherited attribute down the parse tree. Grammar 4.3 presents Grammar 4.2 in Happy, using unevaluated functions and lambda abstractions to evaluate the inherited attribute.

```
Exp  : Exp '+' Exp  { \inher -> f($1 g(inher, $1 inher, $3 inher),
                    $3 h(inher, $1 inher, $3 inher),
                    inher) }
    | int           { \inher -> i($1.val, inher) }
```

Grammar 4.3: Arbitrary Inherited Attributes in Happy

For simplicity, Grammar 4.3 only dealt with one inherited and one synthesized attribute, however, extending this method to multiple attributes is quite simple. The function from one inherited attribute to one synthesized attribute is replaced by a function from all the inherited attributes to all the synthesized attributes. Since Haskell, like most languages, cannot return multiple attributes, a new type will be defined in the manner presented in

Section 4.1. Figure 4.4 shows the data type that would be returned by the function and Grammar 4.4 presents the `Exp '+' Exp` section of Grammar 4.3 with multiple synthesized and inherited attributes. The important syntactic changes are, like in Section 4.1, the `ExpSynth` data type must be constructed (using the `ExpSynth {synth1 = f1(...), synth2 = f2(...), ...}` notation) and that now each synthesized attribute must be accessed using the `synth1 synth2 ...` operators placed in front of the attribute (which, of course, is gotten by passing `$1` all of the inherited attributes).

```
data ExpSynth = ExpSynth {synth1 :: Int, synth2 :: String, synth3 :: Int, ...}
```

Figure 4.4: Data Type for Multiple Synthesized Attributes in Happy

```
Exp : Exp '+' Exp { \inher1 inher2 ... -> ExpSynth { synth1 =
    f1(inher1, ..., g(inher1, ..., synth1 ($1 inher1, ...), ...,
    synth2 ($1 inher1, ...), ...,
    ...
    synth1 ($3 inher1, ...), ...
    synth2 ($3 inher1, ...), ... )
    h(inher1, ..., synth1 ($1 inher1, ...), ...,
    synth2 ($1 inher1, ...), ...,
    ...
    synth1 ($3 inher1, ...), ...
    synth2 ($3 inher1, ...), ... ) )
    synth2 = f2 (...)
    ... } }
```

Grammar 4.4: Multiple Inherited and Synthesized Attributes in Happy

4.4 Generating HERA machine language

To tie all the concepts just presented together and demonstrate the algorithm, consider an example from compiler design. When a piece of code is compiled, the number of registers required must be calculated and then that value is used to determine the most efficient location for each expression to store its result. Grammar 4.5 presents the code for a very simple grammar that only takes plus and parenthesized expressions. This grammar generates HERA machine language code using the Sethi-Ullman approach presented in [1], chapter 9.10, for register allocation.

```

Exp  : Exp '+' Exp      { doPlus $1 $3 }
     | '(' Exp ')'     { $2 }
     | int              { \p -> Node {reg = 1,
                                   code = "SET " ++ show p ++ ", " ++ show $1} }

doPlus expOne expTwo p =
  let addOne = if (reg (expOne p)) == (reg (expTwo p)) then 1 else 0
      thisReg = if (reg (expOne p)) == (reg (expTwo p)) then
                  (reg (expOne p)) + 1
                else
                  max (reg (expOne p)) (reg (expTwo p))
      thisCode = code (expOne ((reg (expOne p)) + addOne)) ++ "\n" ++
                  code (expTwo (reg (expTwo p))) ++ "\n" ++
                  "PLUS " ++ show p ++ ", " ++
                    show ((reg (expOne p)) + addOne) ++ ", " ++
                    show (reg (expTwo p)) in
  Node {reg = thisReg, code = thisCode }

```

Grammar 4.5: Register Allocation in Happy

This thesis is not about register allocation, so the details of the algorithm will not be explained, nevertheless, it is easy to see that there are two named attributes, `reg` and `code`. `reg` is a simple synthesized attribute, passing information about how many registers will be needed in total up the tree. `code` is the inherited-synthesized attribute combination that passes down which register the expression should store its results in, and returns HERA machine language code.

Chapter 5

Conclusion

Evaluation of grammars and inherited attributes provide an extremely important tool in the creation of any compiler. Through a discussion of YACC and Happy it has hopefully become clear that functional languages provide a much larger set of tools for evaluating attributes directly within the grammar, and, although the syntax can be somewhat different and cumbersome at times, functional languages do provide much greater expressive power than their imperative counterparts.

The extra expressive power of functional languages can come at a cost. Functional languages have a larger memory and processing time overhead than their imperative counterparts. Nevertheless, the Haskell language system is extremely well designed, and although identical algorithms will always run faster in C++, Haskell may often be implementing more efficient algorithms than the average C++ programmer will implement. So, like every other problem in computer science, before diving in, it pays to assess what exactly the grammar will be used for as well as the factors that are important in the functionality of the grammar. Once these issues have been resolved, perhaps a functional language will be the logical choice, and hopefully this paper will help guide a programmer through the process of evaluating inherited attributes directly within the grammar.

Bibliography

- [1] Aho, A.V., R. Sethi and J.D. Ullman, "Compilers: Principles, Techniques and Tools", Pearson Higher Education (1985)
- [2] Dornan, C., I. Jones and A. Gill, "Alex User Guide", <http://haskell.org/alex/doc/html/alex.html> (2003)
- [3] Johnsson, T., "Attribute Grammars as a Functional Programming Paradigm", Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture (1987) 154-173,
- [4] Jones, S.P. et al., "Haskell 98 language and libraries: the Revised Report", Journal of Functional Programming, **13** (2003)
- [5] Knuth, D.E., "Semantics of Context-Free Languages", Mathematical Systems Theory, **2** (1968) 127-46
- [6] Marlow, S. and A. Gill, "Happy User Guide", <http://haskell.org/happy/doc/html/happy.html> (2004)
- [7] Turner, D.A., "SNACC: a parser generator for use with Miranda", Proceedings of the 1996 ACM symposium on Applied Computing (1996) 401 - 407