CS 297 Report

By Yan Yao

Introduction

In recent years there has been an active research area which tries to give functional characterizations of time-bounded or space bounded complexity classes. An old example of this would be Cobham's [C64] characterization of PTIME using simple base functions on the natural numbers and closure under composition and bounded recursion on notation. Unfortunately, this characterization is restricted to natural numbers and thus makes it somewhat awkward to implement ``naturally'' occurring algorithms over other structures such as trees. Another avenue of research in this area, has been to give characterizations of these complexity classes which do not explicitly mention the resource bound in question. So for example, one would try to give a functional characterization of polynomial time that does not explicitly mention polynomials anywhere. Such characterizations have been given by Bellantoni-Cook [BC92] and Leivant [L93] under the names of safe recursion and tiered recursion. Still, both safe recursion and tiered recursion have disadvantages in that many naturally occurring polynomial time algorithms do not fit into the rather rigid discipline they impose. Recently, Hofmann [H02] gives a characterization of EXPTIME which allows computations over a variety of data structures such as lists or trees besides natural number. It also does not explictly mention resource bounds. Thus, if a programming language were developed out of this characterization it would give the user the illusion of being able to write code of arbitrary time complexity. Hofmann indicates that replacing general recursion with structural recursion reduces the strength of his system to polynomial time.

My Master's goal is to come up with a subset/variant of C++ guaranteed to be polynomial time. The reason why we are choosing a variant of C++ is that it is a language familiar to many programmers, so the learning curve would be quicker than for an arbitrary functional language.

The idea would be to add a keyword to C++ called ``polynomial'' which one could tag functions or member functions with. the idea is such member function promise to run in polynomial time in their input parameters. We would write a translator from our ``feasible C++'' to regular C++ that translates code not contained in such a polynomial block verbatim into our target. For code in a polynomial block, the code will be parsed to make sure it satisfies the constraints of feasible C++ and, if so, the code will be converted to usual C++. The point of this is that if the translator successfully translate such functions without errors, it would then have verified that the algorithm is in fact a polynomial in its input parameters. The hope is that having such a verifier would help programmers to catch errors in implementing many common data structure algorithms.

To gain knowledge and experience to compelete my project, I have done four reading and programming assignments this semester. The following is the brief introduction of those four deliverables:

- Deliverable #1. The point of the first deliverable is to get practical experience with some of the concepts from theory. I wrote a C program which simulates a Turing Machine program which starts with two numbers a,b separated by a dollar sign (user can supply these number from the command line) on its input and outputs their product.
- Deliverable #2. I read the book “Lex and Yacc”, and used Lex and Yacc to make a C program which can parse the ShML language of Professor Pollett's Fall '01 CS146 classes HW 2. The point of this is to get familiar with lex and yacc.
- Deliverable #3. I Read the book “Computation complexity” and the book “ The C++ Programming Language”. ``Feasible" functions/member functions for C++ were identified . I tried to use notion of safe versus unsafe variables to do this. Professor Pollett has woked together to show correctness of this subset.
- Deliverable #4. I read the paper “Computation Models and Function Algebras “. I Wrote a translator that recognizes the polynomial keyword as suggested above.

Description of the four Deliverables

1 Description of Deliverable #1: Turing Machine

The purpose of my first deliverable is to write a C program which simulates a Turing Machine program which takes two numbers provided by user from the command line and outputs their product. Another purpose was to get practical experience with some concepts of the theory.

1.1 Turing Machine Basics

Viewed as a programming language, the Turing machine has a single data structure, and rather primitive one at that: A string of symbols. The head of the Turing machine is allowed to move to the left and right on the string, to write on the current position, and to branch depending on the value of the current symbol [Pap94]. Formally, a Turing machine is a quadruple $M = (K, \Sigma, \delta, s)$. K is a finite set of states; $s \in K$ is the initial state. Σ is a finite set of symbols. Σ contains the special symbols which represents the blank and the first symbol respectively. δ is a transition function which maps $K \times \Sigma$ to $(K \cup \{h,$ "yes", "no"$\}) \times \Sigma \times \{ \leftarrow, \rightarrow, -\}$.

Initially the state is s. The string is initialized to a Δ, followed by a finitely string $x \in (\Sigma - \{$ blank$\})^*$. X is the input of the Turing machine. The head is pointing to the first symbol. The machine takes every step according to the transition function, changing its state, printing a symbol, and moving the head.

1.2 Simulation of the Turing machine

This C program simulates the Turing Machine which has two tapes. The first tape contains the two input numbers separated by a dollar sign, and the second tape is used to write the calculation result. The program takes two numbers which are provided by user from the command line, and translates them into unary format. Those unary numbers are saved in the first tape, separated by a dollar sign. At the end of the numbers, we put a '0' to indicate the end of the input. For example, 3 * 2 will be represented in the tape as:

1 1 1 $ 1 1 0

This program simulates the Turing Machine to calculate the products of the two numbers. The tape head reads the first symbol. If it is a dollar sign, that means one of numbers equals zero. If not, the tape head moves to the end of the input which signaled by a '0', and moves to the left. It also means one of the numbers equals zero if the '0' is followed immediately by a dollar sign. So the program gives a blank output and terminates when meets those two conditions. If none of the two numbers is zero, the program begins to do the multiply. The steps are as follows:

1) Read the first '1', changes it into dollar sign.
2) Move to the end of the input which is a '0'.
3) Move to the left.
4) If reading a '1' write a '1' on the second tape. (output tape)
5) If it is a dollar sign, move to the left.

6) If it is a dollar sign again, the calculation ends.

The program repeats the above steps until the calculation is ended.

1.2.1 The sample output

Please give two integers: 4, 6
The symbols on the input tape are:
1 1 1 1 $ 1 1 1 1 1 1

The symbols on the output tape are (total number of '1' is 24):
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

2. Description of Deliverable #2: Lex and Yacc

The purpose of deliverable # 2 is to use lex and yacc to make a C program which can parse the ShML language of Professor Pollett's Fall '01 CS146 classes HW 2 and output appropriate shapes. The point of this deliverable is to get familiar with lex and yacc.

2.1 ShML Language

ShML is a language Professor Pollett invented for his CS146 homework. Similar to HTML, ShML has three pairs of tags:

<square>

</square>

<circle>

</circle>

<triangle>

</triangle>

The meaning of a pair of tags is to draw that kind of shape in the current enclosing shape. If there is no enclosing shape the shape should be drawn directly on the frame's background. For example, if the ShML file was the following:

<square>

<circle>

</circle>

<triangle>

</triangle>

</square>

<square>

</square>

then two non-overlapping squares would be drawn centered on the main frame. In the interior of the first square, would be drawn in a non-overlapping manner a circle and triangle.

An example of a file with mismatched tags is:

<square>

<circle>

</circle>

</triangle>

</square>

As another example of what an ShML image might look like the file:

<triangle>

<circle>

<square>

<circle>

</circle>

</square>

</circle>

<square>

</square>

<circle>

</circle>

</triangle>

2.2 Using Lex

2.2.1 Basic Specifications

The general format of Lex document is:

```
{definitions}
```

```
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. For example, the rule

```
integer   printf("found keyword INT");
```

to look for the string integer in the input stream and print the message ``found keyword INT" whenever it appears. The end of the expression is indicated by the first blank or tab character.

2.2.2 Regular Expressions

Lex takes a set of descriptions of possible tokens and produces C routine, which we call a lexer, or a scanner which can identify those tokens.The token descriptions that lex uses are known as regular expressions. The following is a list of examples of some regular expressions:

- A regular expression specifies a set of strings to be matched. It contains *text characters* and *operator characters*.
  - The operator characters are the following:

    ```
    "  \  [  ]  ^ -  ?  .  *  +  |  (  )  $  /  {  }  %  <  >
    ```

  - Everything else is a text character.

- An operator character may be turned into a text character by enclosing it in quotes, or by preceding it with a \ (backslash).

- Matching multiple occurences of characters:
  - * matches *zero* or more occurences of a
  - + matches *one* or more occurences of a
  - {m,n} matches m through n occurences of a .

- Checking for context:
  - a/b matches "a" but only if followed by b (the b *is not* matched).
  - a$ matches "a" only if "a" occurs at the end of a line (i.e. right before a newline). The newline is not matched.

    - ^a matches "a" only if "a" occurs at the beginning of a line (i.e right after a newline).

- Sets of characters
    - [abc] matches any charcter that is an "a", "b" or "c" (and nothing else).
    - [a-zA-Z0-9] Matches any letter (uppercase or lowercase) or digit.
    - [^abc] matches any charcter *but* "a", "b" and "c". Note the two uses of "^" for context as well as forming the complement.

- Miscellaneous:
    - Grouping: Use () to group expressions
    - Optional arguments: e.g. ab?c matches abc and ac
    - Blanks: can only be matched by putting a blank inside quotes.
    - Tabs: Either by pressing the key between quotes, or by its escape sequence "\t"
    - Newlines: By its escape sequence "\n".
    - Default: the "." matches *everything* except a newline.

- Within the square brackets most operators lose their special meanings. The exceptions are: "\" and "-". The "^" loses its usual meaning but takes on a new one.

- "\n" *always* matches newline, with or without the quotes. If you want to match the character "\" followed by "n", use \\n .

2.2.3 Lex Actions

When an expression written as above is matched, Lex executes the corresponding action.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like [a-z]+. Lex leaves this text in an external character array named yytext. Thus, to print the name found, a rule like

```
[a-z]+   printf("%s", yytext);
```

will print the string in yytext. The C function printf accepts a format argument and data to be printed; in this case, the format is ``print string" (% indicating data conversion, and s indicating string type), and the data are the characters in yytext. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+   ECHO;
```

is the same as the above.

2.3 Using Yacc

2.3.1 Basic Specifications

Every yacc specification file consists of three sections: the definitions, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.) A full specification file looks like

definition
%%
rules
%%
programs

The definition section includes declaration of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can include a literal block, C code enclosed in %{ %} lines. The rule section consists of a list of grammar rules. A colon is put between the left and right hand sides of a rule, and a semicolon is put at the end of each rule.

2.3.2 Yacc Grammars

Yacc takes a concise description of a grammar and produce a C routine that can parse that grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar. A grammar is a series of rules that the parser uses to recognize syntactically valid input. Yacc grammar rule has the following general form:

result: *components*...
;

where *result* is the nonterminal symbol that this rule describes and *components* are various terminal and nonterminal symbols that are put together by this rule .

Usually there is only one action and it follows the components. Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character `|' as follows:

```
result:    rule1-components...
        | rule2-components...
        ...
        ;
```

They are still considered distinct rules even when joined in this way. If *components* in a rule is empty, it means that *result* can match the empty string.

2.3.3 Recursive Rules

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. This important ability makes it possible to parse arbitrarily long input sequences. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this *left recursion*. By contrast, here the same construct is defined using *right recursion*:

```
expseq1:  exp
        | exp ',' expseq1 ;
```

2.4 Using Lex and Yacc in Deliverable #2

Lex and yacc are tools designed to transform structured input. Any application that looks for patterns in its input, or has an input or command language is a good candidate for lex and yacc. In programs with structure input, the lexer divides the input into meaningful units, the parser discovers the relationship among the units (tokens).

2.4.1 Description of The Program

This program consists of three parts:

- test.txt: source code for lex part. The purpose of this part is to read ShML file, then identify all the tokens. Tokens are valid tags: <circle>, </circle>, <triangle>, </triangle>, <square>, </square>.

- test.h: data structures header file.

- test.y : source code for yacc part. In this part, we define tokens and the grammar rules. The rules are recursive as following:
    - start: Triangle start| Circle start | Square start | ;
    - Triangle: TRIANGLE_START start TRIANGLE_STOP;
    - Circle: CIRCLE_START start CIRCLE_STOP;
    - Square: SQUARE_START start SQUARE_STOP;

Due to the lack of graphics library in gcc, we did not implement the graphics as the final output. Instead of this, we output the structure to if it matches the input file.

3 Description of Deliverable #3: Restrictions

The task of deliveranble #3 is to read the book "Computation complexity" and the book " The C++ Programming Language", then identify ``feasible" functions/member functions for C++. Finally I was supposed to give proper restrictions to certain C++ functions/member functions, thus, guarantee those functions can be computed in polynomial time.

3.1 Complexity Classes

A complexity class is specified by several parameters. First, a complexity class is characterized by a mode of computation. The most important modes are known as deterministic mode and nondeterministic mode. Second, the basic resources of a complexity class are time and space. Finally, we must specify a bound which is a function f that maps nonnegative intergers to nonnegative integers. The complexity class is defined as the set of language decided by some Turing machine M operating in a mode, and such that, for any input x, M expends at most f(|x|) units of the specified resources.

3.2 Restrictions

I gave a set of restrictions to certain C++ functions/member functions to guarantee those functions are polynomial.

3.2.1 For loop: for (for-init statement condition; expression) statement

The for-statement is intended to express fairly regular loops. A for loop usually consists of loop variable, the termination condition, and the expression that updates the loop variable. We gave following restrictions to guarantee the for loop can be performed in polynomial time.

- If no initialization is needed, the initializing statement can be empty. If the condition is empty, the for-statement will loop forever unless it exits by a break, return, goto. A bad example:

```
for (    ;         ;         )
{
    expressions
}
```

- If the expression is omitted, we must make sure if the loop variable is updated in the body of the loop. For example:

```
for (i = 0; i <= 10; )
{
        i ++;
}
```

- If the loop variable is updated in the expression, it shouldn't be updated again in the body of the loop. A obvious mistake can be made as this exmpale:

```
for (i = 0; i <= 10; i++)
{
        i --;
}
```

- If the comparison operator in the condition is ">=" or ">", then the loop variable should be incremented. If the comparison operator is "<=" or "<", then the loop variable should be decremented.

- Loop variables are not allowed to be assigned to a pointer. And, pointers are not allowed to be used as loop variables.

3.2.2 While loop: while (condition) statement
Do statement while (expression)
A while-statement simply executes its controlled statement until its condition becomes false. In this case, we set a loop counter to limit the loops. Once the loop counter exceeds a fixed value, the program will be terminated.

3.2.3 Switch
The switch-statement tests the value of its condition, which is supplied in parentheses after the switch keyword. The break-statement are used to exit the switch statement. The constants following the case labels must be distinct. break is the most common way terminate a case. A continue statement can also be used to go to the very end of a loop statement.

3.2.3 Goto : goto identifier;
Identifier: statement
The scope of a label is the function it is in. You can use goto to jump both into and out of blocks. The only restriction is that you cannot jump past an initializer or into an exception. In my project we disallowed goto.

3.2.4 Recursion: undecided.

3.2.5 Function calls

Functions which are called in the polynomial block should also be checked to see if they have polynomial tag. We should label the function with polynomial tag.

3.2.6 Side-Effects

The standard library is guaranteed to be polynomial time.

3.2.7 Function Prototypes

If the function prototype is labeled with polynomial time, we should also check the function to make sure if it is polynomial time. If the function is not defined in the source file then give error message.

4. Description of Deliverable #4

The main task of deliverable #4 was to write a translator using lex and yacc. The translator can parse a valid C++ file and can recognize the polynomial keyword in this file.

4.1 Polynomial Tag

A polynomial tag “polynomial” was introduced into my project. The idea of the tag is that if a polynomial tag is put in front a function, we shall apply our restrictions to this function to see if it is a polynomial function. The polynomial tag can be placed at any part of the file. However, it must be placed in front of valid function. So it is very important to recognize a valid polynomial tag. In deliverable #4, due to the lack of time I only developed a translator to recognize the polynomial key word in a C++ source file.

4.2 Implementation

The implementation of this translator can be divided into two parts: lexer part and parser part. The lexer reads the source file and returns the tokens to the parser; the parser has some rules to check if this is a valid polynomial function.

4.2.1 Tokens

In a C++ source file, code can be recognized as several main groups.

- Preprocessors: if an input file has a “#” at the beginning of the file, we ignore the whole line;
- Data/function type: int, float, char, double, void, bool ;
- Special symbols: “(“, “)”, “{“, “}”, “;” ;
- Polynomial tag: polynomial;
- Function/variable name: any text except the key word we stated above.

Those key words will be return to the parser as tokens. I have not defined the tokens that appear as function parameters and contents inside the function body. I simplified this step by applying any texts inside the ( ) or {} as valid contents.

4.3.1 Rules

The parser takes the tokens and feeds them into the rules. In this simple translator, I gave some rules to identify basic structure of a C++ source file and the polynomial function. A source file mainly consists of five elements: preprocessors, declaration, function, function

prototype and polynomial function. Please note that this is just a simplified definition of C++ source file.

- Preprocessor: if a "#" is met, we consider this is a preprocessor;

- Declaration: a declaration consists of a data type, variable name and a semicolon. It's format should be like this:

  Data type variable name ;

- Function prototype: a function prototype's format :

  Function type function name (anything) ;

- Function: a function consists of a function type, function name, parameters and function body.

  ```
  Function type  function name ( anything)
  {
      anything
  }
  ```

- Polynomial function: a polynomial function is a polynomial tag followed by a valid function:

  ```
  Polynomial  Function type  function name ( anything)
              {
                  anything
              }
  ```

5 Conclusions

My master project is to give a set of restrictions to functions/member functions in C++, and develop a translator to apply those restrictions to the polynomial functions in any C++ source file to see if the polynomial functions can be complete in polynomial time. A polynomial tag was introduced in my project. If a function follows a polynomial tag, we will check this function to see if it is a polynomial function. This translator is like a preprocessor that can help C++ programmers to find mistakes in common implementing data structures and algorithms.

This semester I have done four deliverables as I described above. The first one gave me concepts of how Turing machine works. The second deliverable gave me experience working with lex and yacc. The third and fourth deliverable combined the concepts of polynomial computation and compiler designing together. Therefore, I was able to write a simple translator to recognize the polynomial key word in my self-defined C++ source file. The deliverable #4 was only a simple test of my theory and practice study of my CS297.

Further work will be done in CS 298. Next semester, I will continue to give more strict definitions of the C++ source file and continue to verify the correctness of the rules. A more advanced translator will be complete at the end of CS 298.