# CSCI 3250/5250 - Compilers
# Class Notes
# Spring 2022

David Ferry

2022/03/25
13:01:47

# Contents

# Class 0

## 0.1 Why take Compilers as a course?

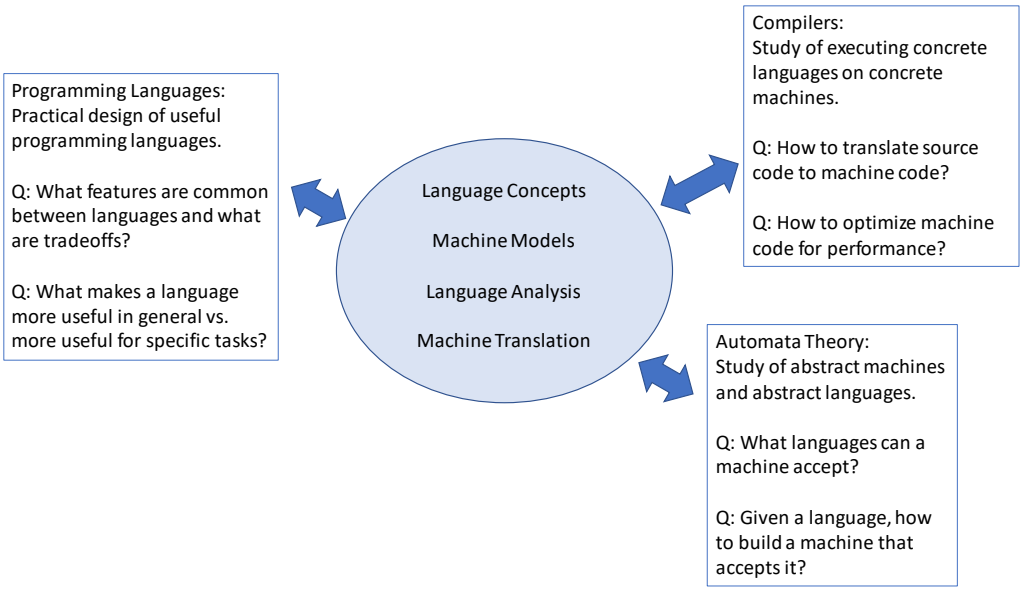



Figure 1

## 0.2 What is a Compiler

Basic Understanding of what a compiler does:

Source Code ⇒ Compiler ⇒ Machine Code

Source Code ⇒ Interpreter ⇒ Interpreter Runtime

Source Code ⇒ Bytecode ⇒ Virtual Machine

There are typically interpreted languages, such as Python or JavaScript, and there are

languages that are both interpreted and compiled, such as Java. Technically you can build compilers for languages like Python and JavaScript, but these tend not to be used in practice, or are implemented as Just-In-Time (JIT) compilers.

In reality, the process has many, some possibly optional, steps. See Fig 1.1 on pp 4. These include, among others:

- Lexical analysis (tokenization)
- Syntax analysis (grammar)
- Semantic analysis (meaning)
- Translation to an intermediate representation
- Optimization
- Target Code Generation

Conceptually, we can think of each step as a single language transformation. We're not transforming source code into machine code, each step transforms from a source language to a target language. The whole compilation process is a chain of individual translation steps.

An example from the Dragon book is in figure 2 on page 6.

From a practical perspective, equally important is how the overall process is modularized.

- Some intermediate stages are optional
- Some stages are specific to a single language (lexing, parsing)
- Some stages are common to multiple languages (optimization of an intermediate representation)
- Some stages are specific to single output targets (target code generation)

We don't want to re-build the entire compiler over and over again for every individual language pair (e.g. C to x86, C to ARM, Java to x86, Java to ARM). Loosely, most modern compilers are structured with a front end, middle, and back-end with this in mind.

We will talk about most of these modules individually, and we will talk about the interfaces and data structures we use to pass data between them as well.

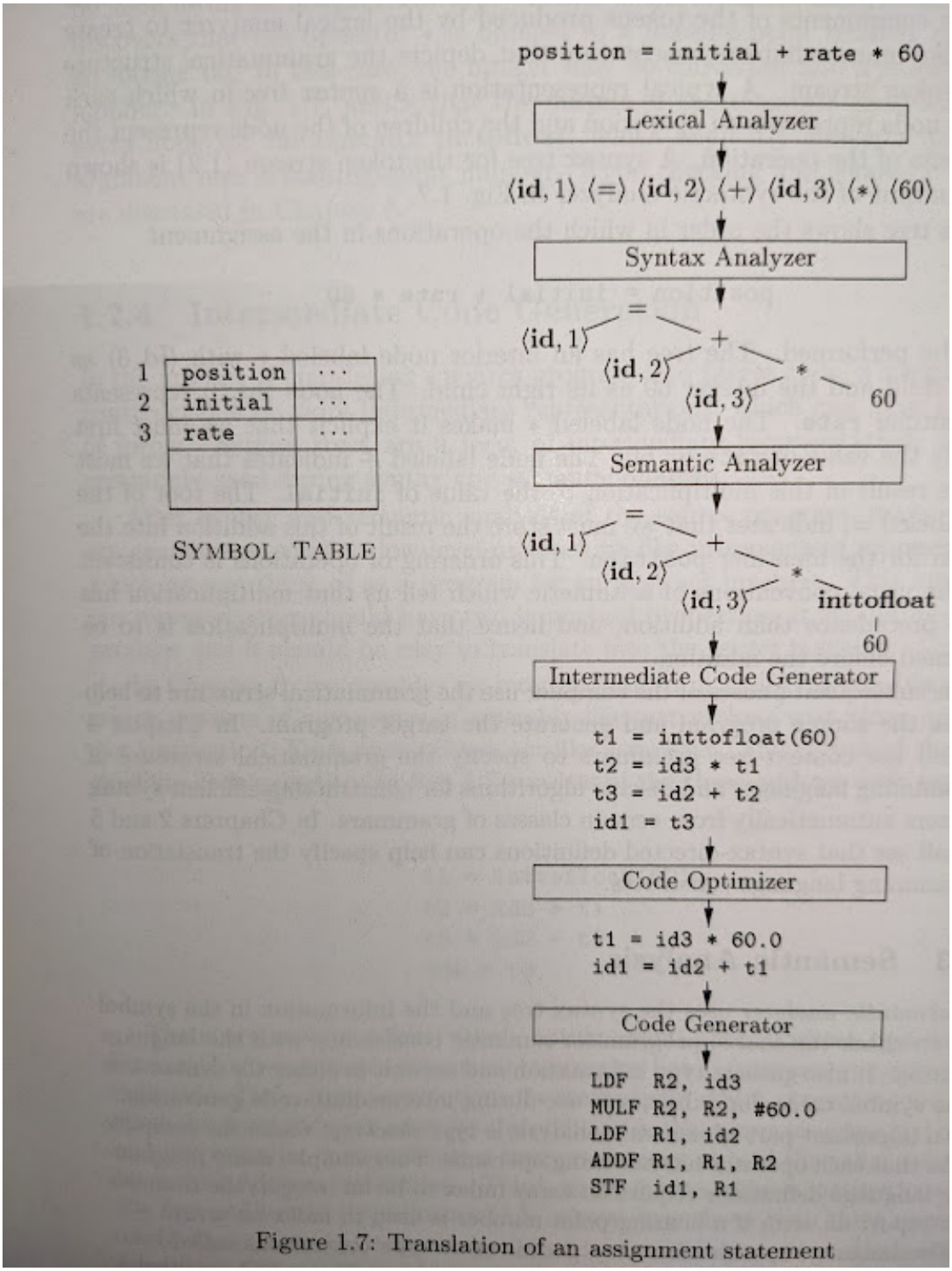

position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position ···
2 initial ···
3 rate ···
SYMBOL TABLE
Figure 1.7: Translation of an assignment statement

Figure 2

## 0.3 Using the Linux System at SLU

The CS department provides a shared Linux system called Hopper. This system is the officially supported way for you to access the SML interpreter/compiler, and it is where I will grade your work. You are free to set up SML on your own machine, but please verify that your code runs on Hopper before submission.

By virtue of being enrolled in a CS course, you should have had an account generated for you. If this is your first time in a CS course, you should have had an automatically generated password sent to you via an email. This system provides terminal-only (non-graphical) access. If you have never used the Linux command line before, you will become comfortable with it by the end of this course.

You will need an SSH client to connect to Hopper. Linux and Unix-derived systems (including Apple/Mac) usually have a terminal-based SSH client already built in. There are a variety of alternative SSH clients for all kinds of systems, including Windows. My favorite is a Chromium/Chrome browser extension called Secure Shell. (If you have only ever used PuTTy, you owe it to yourself to explore some alternatives!)

The standard way to connect via terminal SSH client, after replacing username with your actual username, is:

```
ssh username@hopper.slu.edu
```

After which you will be prompted for your password. If you are connecting via a graphical interface, the Hostname or Servername is hopper.slu.edu, the username is your actual username, and the port number should be port 22.

If you are not familiar with the Linux terminal, please spend some time becoming proficient in creating/navigating directories and the filesystem, and creating files and editing files with a text editor. My favorite text editor is VI/VIM, but there is a steep learning curve. Many students use Nano, Emacs, VI/VIM, and others. If you ever get stuck, a web search is always the best way to get unstuck, such as "How do I save and quit in Emacs?"

Some selected, hopefully helpful, example commands for command line first timers are in Table 1 on page 8.

| | |
|---|---|
| man <command> | Display the manual page (help page) for a command. Use arrow keys and PageUp/PageDown to navigate. Press q to quit. |
| man man | Display the help page for the help pages. |
| ls | Show a list of files and directories in the current directory. |
| ls -lah | Show a list of files and directories in the long-listing format, including all hidden files/directories, and using human-readable sizes |
| cd <directory> | Move into the specified directory. |
| cd .. | Move up one directory from where you are now. |
| cd ~ | Move to your home (starting) directory. |
| mkdir <new directory> | Create a new directory. |
| touch <new file name> | Create a new, empty file. |
| vi <file name> | Open the specified file with VIM, or create a new file if it doesn't exist. |
| emacs <file name> | Open the specified file with Emacs, or create a new file if it doesn't exist. |
| nano <file name> | Open the specified file with Nano, or create a new file if it doesn't exist. |
| cp <source> <destination> | Copy the source files to the destination, omitting directories. |
| cp -r <source> <destination> | Copy the source files to the destination, including directories. |
| mv <source> <destination> | Move the source files and directories to the destination. |
| rm <file> | Delete the specified files, skipping directories. (Note: This is permanent!) |
| rm -r <name> | Delete the specified files and directories. (Note: This is permanent!) |

Table 1: Helpful example commands for command line first timers.

# Class 1

## 1.1 Functional Programming

Our book author claims that functional programming is a natural setting to describe and write a compiler in. We will have to take him at his word for now!

Functional programming is one of a variety of paradigms or styles of programming:

- Imperative (assembly code, GOTO, compare and jump)
- Structured (loops, if-then-else blocks, switch statements, etc.)
- Procedural (emphasis is on writing modular procedures or functions that can be re-used)
- Object-Oriented (emphasis is on objects)
- Functional

So what is a functional language? I am not a programming languages expert, but loosely:

- Encourages functional style
- Encourages *evaluation of expressions* rather than *execution of a sequence of commands*
- Encourages "stateless" programming, discourages use of variables, modifying memory, and other "side-effects"

## 1.2 ML, the *Meta-Language*

Our functional programming language for this course is ML, a.k.a. the *Meta-Language*. ML is an earlier functional language, and has grown over time. There are several flavors of ML, several implementations and compilers/interpreters to use. I will be using the *Standard ML of New Jersey* or *SML* tools for this course. This is free, open-source

software. I have asked for it to be installed on our department systems. You can also build from the source code, which is gotten here:

```
https://www.smlnj.org/
```

Note: These notes are not a well-rounded introduction or adequate reference for ML. Many other documents exist, both online and in book form, both free and paid- I would suggest finding some of those as your "desk reference" for you to use during this class. The course syllabus has pointers to a few such (free) documents I have found helpful and well written.

With that said, here are some specific language features of SML:

- SML is a functional language
- SML is both interpreted and compiled
- SML programs (mostly) have no variables or internal state. It does have *named values*
- SML is strongly typed! Type rules are absolute and ML will never implicitly cast a value of one type to another type
- SML is highly polymorphic. It's easy to write functions that operate on abstract data types
- SML functions are flexible. Functions are easily created on the fly and can be passed into other functions as arguments

## 1.3 Installing SML-NJ

If you want to install SML-NJ on your own machine, there are instructions for Linux, Windows, and Mac. Refer to the official SML-NJ page at:

```
https://www.smlnj.org/
```

If you are running Ubuntu Linux or similar, you can use the commands:

```
sudo apt install sml
sudo apt install ml-lex
sudo apt install ml-yacc
```

Otherwise, you can build SML-NJ from source. Talk to the instructor for details.

## 1.4 Running SML Code

In this class, we will typically either run code interactively in the SML interpreter, or we will write a source code file and pass that file to the interpreter. The SML-NJ interpreter is just called `sml`. When you run SML you will get a prompt indicating the interpreter is ready to execute. For example:

```
dferry@mobius:~$ sml
Standard ML of New Jersey v110.79 [built: Sat Oct 26 12:27:04 2019]
-
```

The dash indicates that the interpreter is ready for a new line of input. For example, we can enter "5+10" and get:

```
- 5 + 10;
val it = 15 : int
-
```

Before we go any farther, let me say up front that the standard way to quit the interpreter is with the CTRL-D combination. This sends the "end of file" character from the keyboard and causes the interpreter to quit. The CTRL-C combination typically sends an interrupt and quits a program, but that behavior is overridden in SML-NJ to instead just abort the current input.

Note that the semicolon terminates the statement and causes the interpreter to execute the code. If you don't include the semicolon, the interpreter assumes you're going to give it more input. This is noted with an equals sign instead of the single dash:

```
- 20 + 30
= ;
val it = 50 : int
-
```

As mentioned above, you can use source code files to hold more complicated code, or a sequence of code statements that you might not want to re-type over and over again. If I have a file called `code.sml` with the following statements:

Program `code.sml`

```
1  val a = 10;
2  val b = 20;
3  val c = a + b;
```

Then, we can execute this file with the "use" command in the interpreter:

```
Standard ML of New Jersey v110.79 [built: Sat Oct 26 12:27:04 2019]
- use "code.sml";
[opening code.sml]
val a = 10 : int
val b = 20 : int
val c = 30 : int
val it = () : unit
```

## 1.5 Values and Basic Expressions

A standard ML program usually has a set of declarations of *values*, followed by some function invocations that cause some processing to happen. Here we define a variety of types:

Program `expressions.sml`

```
1  val flag = true;
2  val pi = 3.14159;
3  val radius = 2;
4  val characterD = #"d";
5  val name = "David Ferry";
6
7  val square = fn x => x*x;
8  fun double y = 2*y;
9
10 val squared = square 3;
11 val doubled = double 5;
```

Which when executed causes the following output:

```
- use "expressions.sml";
[opening intro1.sml]
val flag = true : bool
val pi = 3.14159 : real
val radius = 2 : int
val characterD = #"d" : char
val name = "David Ferry" : string
val square = fn : int -> int
val double = fn : int -> int
```

```
val squared = 9 : int
val doubled = 10 : int
```

Notice that SML has a strong and explicit type system. Each value is given a type as it is interpreted, and these types are reported to the user.

Let's also notice lines 7 and 8 of this program, both defining a function. These two forms are equivalent, with the `fun` keyword on line 8 being a shorthand for what is written on line 7. See also in the output of these two lines- both `square` and `double` are considered regular values in SML. Functions may be defined and passed as arguments to other functions exactly as regular values are. Notice also that functions operate on the token to their right- they do not need parentheses to be invoked.

## 1.6 Class 1 Exercises

Think of these exercises as guided learning. Our class time and this document will not have prepared you to answer all these questions, you are encouraged and expected to use your knowledge of programming and other resources to solve them.

1.6.0.1. Successfully define the following high level, plainly stated values in the SML interpreter:

- a = 5
- e = 2.71828
- b = the negation of the first item in this section
- c = the character C
- n = a string containing your name
- f = the boolean value false

1.6.0.2. Successfully evaluate the following high level, plainly stated expressions in the SML interpreter:

- 5 + 7
- 5 - 7
- 10 + $\pi$
- 5 $\times$ 3
- 5 $\times$ 2.5
- Three and a half plus two and a half
- 16 / 8 = 2

- 13 / 4 = 3 (integer division)
- 13/4 = 3.25 (real division)

1.6.0.3. Accomplish the following using the SML interpreter:

- Decode the ASCII sequence 83 76 85 using the “chr” function
- Convert “Hello” into ASCII by using the “ord” function
- Define strings `first` and `last` that contain your first and last name, respectively
- Concatenate those strings together with a space between for your full name
- Determine the length of the string from the last item with the “size” function

1.6.0.4. Successfully evaluate the following expressions to a boolean value in the SML interpreter:

- 5 is less than 10
- 5 is less than -7
- $\pi$ is less than 10
- $\pi$ is less than or equal to 10
- 12 is equal to 12
- 12 is equal to 24
- 12 is not equal to 24
- character a is less than character A
- string “david” is less than “JOE”
- string “DAVID” is less than “joe”
- string “DAVID” is less than “JOE”
- strings “ABC” and “DEF” are equal
- strings “ABC” and “ABC” are equal

# Class 2

## 2.1 Functions

Functional languages, and in particular the functional programming mindset, can be a difficult adjustment from traditional structured programming. There is no better place to contrast this attitude than the approach of such languages to the concept of functions. While a structured language provides functions as a form of code re-use and data encapsulation, functional languages use the concepts of functions and function composition as the basic method for a programmer to order their thoughts.

A function would be defined in a basic mathematics textbook as follows:

$$f : X \to Y$$

A function $f$ is a mapping of values from the *domain* $X$ to the *codomain* $Y$. A specific element of the domain $x \in X$ maps to a specific element $y \in Y$ called the *image* of $x$ under $f$. This is denoted as:

$$f(x) = y$$

Functions can be described as an explicit mapping between values, and we need not restrict ourselves to numerical functions:

$$(f : Colors \to Vehicles) \Rightarrow \left( \begin{array}{c|c} x & f(x) \\ \hline \text{red} & \text{boat} \\ \text{green} & \text{truck} \\ \text{blue} & \text{car} \end{array} \right)$$

However, specific functions where the domain and codomain are numbers will often be described with an equation, such as the following two:

$$f(x) = 2x$$

$$g(x) = x^2$$

The heart of the functional style of programming is *function composition*. As we build more and more complex functions, it may become more and more difficult to express them explicitly and directly. Instead, it may be more convenient to express them as

combinations of other functions. For example, suppose we are interested in the following function:

$$h(x) = 2x^2$$

If for whatever reason it was hard to express the function $h$ directly, notice that we can also express it as a composition of $f$ and $g$:

$$h(x) = f(g(x))$$

Written in SML, this all would look like the following:

Program `functions.sml`

```
1  fun f x = 2*x;
2  fun g x = x*x;
3  fun h x = f( g x );
4
5  f(5);
6  g(5);
7  h(5);
```

The SML function declaration has three parts:

`fun` *Name Parameter* `=` *Expression*`;`

Notice that this is quite unlike most non-functional programming languages. First, the entire body of the function is the return value of the function. In structured programming languages, functions are typically a list of statements to execute. Second, the function takes exactly one input argument, while most other languages allow many parameters. (However, that single input parameter may be a tuple or list of multiple values!)

The above program produces the following output:

```
- use "functions.sml";
[opening functions.sml]
val f = fn : int -> int
val g = fn : int -> int
val h = fn : int -> int
val it = 10 : int
val it = 25 : int
val it = 50 : int
```

## 2.2 If-Then Statements

SML supports conditional statements with the following syntax:

```
if boolean_expression
then expression_1
else expression_2
```

If `boolean_expression` evaluates to true, then the first expression is evaluated and returned, otherwise the second will be. For example, the following code will return the character T.

```
if 5 < 10
then #"T"
else #"F"
```

The whitespace and new lines are irrelevant, and only placed there for readability. The following code is equivalent.

```
    if 5 < 10 then #"T" else #"F"
```

SML also supports case statements. In fact, If-Then statements are really just a special form of case statements. A case statement is of the following form:

```
case main_expression
of pattern_1 => expression_1
 | pattern_2 => expression_2
 ...
 | pattern_n => expression_n
```

When the case statement evaluates, it matches `main_expression` to a `pattern`, and returns the associated `expression`. Note that `main_expression` and each `pattern_i` must have the same type, and each `expression_i` must have the same type as well. However, the `pattern` types and the `expression` types do not need to be equal.

An If-Then statement can be written as a case-statement:

```
case boolean_expression
of true  => expression_1
 | false => expression_2
```

An example program using both If-Then statements and Case statements. Note the placement of the semicolons.

Program `conditionals.sml`

```
1  val a = 42.0;
2  val b = 100.0;
3
4  if a > b
5  then "Life, the Universe, and Everything"
6  else "Or not";
7
8  val mode = 4;
9
10 case mode
11 of 1 => "First"
12  | 2 => "Second"
13  | 3 => "Third"
14  | 4 => "Fourth"
15  | 5 => "Fifth"
16  | _ => "Other";
```

Which evaluates as such:

```
- use "conditionals.sml";
[opening conditionals.sml]
val a = 42.0 : real
val b = 100.0 : real
val it = "Or not" : string
val mode = 4 : int
val it = "Fourth" : string
```

## 2.3 Parentheses

Parentheses are used very differently from most common C-like programming languages. They are primarily used for creating tuples and overriding default evaluation orders. A specific contrast to C-like languages is that they are not used for invoking functions. As a learner, especially if you are experienced in a C-like language, I suggest that you only add parentheses when you think or confirm they are necessary. Many times the default position in C-like languages is to add extra parentheses to clarify, such as when making order of operations explicit. Resist the temptation!

Program `intro2.sml`

```
1  val square = fn x => x*x;
2  fun double y = 2*y;
3
4  val result  = square 5;
5  val result2 = double 20;
6  val result3 = square( double 5 );
```

Yields the output:

```
- use "intro2.sml";
[opening intro2.sml]
val square = fn : int -> int
val double = fn : int -> int
val result = 25 : int
val result2 = 40 : int
val result3 = 100 : int
```

Notice that parentheses are not required for invoking the functions `square` or `double`. However, the parentheses on line six are strictly required! Remember that functions in SML are considered values, and can be passed to functions like regular values. Executing the line without parentheses results in an error:

```
- square double 5;
stdIn:26.1-26.16 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         int -> int
  in expression:
    square double
```

Like every interpreter or compiler, SML tells you exactly what is wrong with excruciating detail and typically in the least helpful way possible. The error is "operator and operand don't agree" and "tycon mismatch" means that the disagreement is a type error. In this case, the function `square` is the operator, and this function expects an integer, giving the statement "operator domain: int". Written without parentheses, the function `square` operates on the token immediately to its right, which is the function `double`. The operand in this case, `double`, is a function that takes an integer and returns an integer, hence the message "operand: int => int". Helpfully, the interpreter does isolate the error to the specific expression "square double" instead of the entire line "square double 5".

Trying to call a two-argument function with parentheses can get you into trouble. The following two function definitions are not identical:

```
fun addTwo x y = x + y;
fun addTwo (x, y) = x + y;
```

The first declaration accepts two integers, while the second declaration accepts a tuple of two integers. Under SML's type system these are not the same! Note the type signatures returned after the definition of each function below.

```
- fun addTwo x y = x + y;
val addTwo = fn : int -> int -> int
- addTwo 5 10;
val it = 15 : int

- fun addTwo (x, y) = x + y;
val addTwo = fn : int * int -> int
- addTwo (2, 4);
val it = 6 : int
```

## 2.4 Functions, Case Statements, and a New Function Syntax

We've already made something of a big deal about how functions are considered first class objects in SML. Now, we'll make an argument that *everything* in SML can be considered a function, in a sense. While this might seem esoteric or academic, it's actually quite relevant in some fields, such as formal program verification.

Let's make a function that implements the exclusive-OR operator. Given two boolean inputs, this function returns TRUE if exactly one of those inputs is true, but returns false if both inputs are false or if both inputs are true. We can do this using If-Then statements, but it's something of a mouthful. Note that the following function accepts a tuple with two booleans:

Program `XOR.sml`

```
1  fun XOR (x, y) =
2   if x = true
3   then ( if y = true
4          then false
5          else true )
6   else ( if y = true
```

```
7           then true
8           else false );
9
10  XOR ( true,  true  );
11  XOR ( true,  false );
12  XOR ( false, true  );
13  XOR ( false, false );
```

There are multiple ways to write this function correctly. If we were to use a case-statement the code might be slightly more stratightforward:

Program `XORcase.sml`

```
1  fun XOR (x, y) =
2    case (x, y)
3    of ( true,  true  ) => false
4     | ( true,  false ) => true
5     | ( false, true  ) => true
6     | ( false, false ) => false;
```

We saw previously how If-Then statements are really just a special case of the more general case statement, and the transformation between `XOR.sml` and `XORcase.sml` makes sense in that light. Moreover, we will also see that in the functional programming paradigm this "function encapsulating a case statement" becomes a frequently used idiom. To that end, there's actually a special *clausal function* notation that combines these:

Program `XORclausal.sml`

```
1  val XOR = fn
2        ( true,  true  ) => false
3     | ( true,  false ) => true
4     | ( false, true  ) => true
5     | ( false, false ) => false;
```

This is essentially creating a named case statement, but it is considered a function because once we name it and re-use it, it's no longer a single expression but an encapsulated function. Pretty neat, huh? To bring us back full circle to the first sentence of this section, consider that *any case statement* can be implemented as an *anonymous function* (also called a *lambda function* by some) as such. The following program shows exactly that, and in fact the case statement is just a syntatic shortcut for defining and using an anonymous function.

Program `XORanonymous.sml`

```
1  val (x, y) = (true, false);
2
3  case (x,y)
4    of ( true,  true  ) => false
5     | ( true,  false ) => true
6     | ( false, true  ) => true
7     | ( false, false ) => false;
8
9  (fn ( true,  true  ) => false
10  | ( true,  false ) => true
11  | ( false, true  ) => true
12  | ( false, false ) => false) (x,y);
```

## 2.5 Currying and Partial Function Application

We have already stated that functions in SML are always functions of one parameter. However, the language does allow us to make function declarations of the form:

```
fun sum x y = x + y;
```

Which returns the following type information without error:

```
val sum = fn :  int -> int -> int
```

What does that mean? Strictly speaking, `sum` is a function that takes a single integer and returns another function, which itself accepts an integer. In other words, the code:

```
fun sum x y = x + y;
 sum 10 5
```

Will first apply `10` to `sum`, producing an intermediate form:

```
 fun sum 10 y = 10 + y
```

In effect, this function has been written in its *curried* form. Currying is the process of converting a multi-argument function into a sequence of single-argument functions. For example, a function in SML might take a tuple (A, B, C) giving the signature:

```
val uncurried = fn :  ( A * B * C ) -> D
```

However, it can be trivially converted to a curried form:

```
val curried = fn :  A -> B -> C -> D
```

This is not just syntactic difference- the uncurried form can only be called with all three inputs at the same time. The curried form permits us to do *partial function application*, exactly as we have seen above. We partially evaluate the function `sum` with the value `10` and the result is another function, expecting the other argument to sum and returning an integer.

The currying process is automatic and the normally the intermediate form is produced and consumed automatically by the interpreter. However, we can also produce our own intermediate form:

Program `partialApplication.sml`

```
1  fun curriedSum x y = x + y;
2
3  (*Called normally: *)
4  val result = curriedSum 10 5;
5
6  (*Partial Application- return intermediate function explicitly: *)
7  val partial = curriedSum 20;
8  val result = partial 30;
```

Producing the output:

```
- use "partialApplication.sml";
[opening partialApplication.sml]
val curriedSum = fn : int -> int -> int
val result = 15 : int
val partial = fn : int -> int
val result = 50 : int
```

In particular, note that the value `partial` has type:

```
val partial = fn :  int -> int
```

It is itself a function that takes an int and produces an int. Thus, the original definition of `curriedSum` makes sense. SML reports `curriedSum` has type `fn :  int -> int -> int`, but the function `sum` might be described more clearly as `fn :  int -> ( int -> int )` .

The purpose of currying and partial function application is to simplify syntax and function application. For example, a function that computes how long it takes to travel between cities can illustrate this. Consider a function that takes a mode of transportation and two cities, and returns the time between them:

```
val timeBetweenCities = fn :  vehicle -> city -> city -> real
```

We can curry the function to create multiple other useful functions easily:

```
val timeFlyingFromSTL = timeBetweenCities Airplane "St. Louis";
```

Now we have created a new function `timeFlyingFromSTL` effectively for free.

We will see other uses of currying, especially when processing lists and other recursive data structures.

Lastly, there is the obligatory disclaimer: the term *currying* refers to mathematician Haskell Curry, not the food. If you have done functional programming before, you might recognize his first name as well!

## 2.6 Class 2 Exercises

2.6.0.1. Successfully create and execute the following basic functions:

- Write the function `cubed` which computes the cube of the input.
- Write the function `rectPrism` which takes a length, width, and height and produces a tuple ( volume, surfaceArea ). The volume is given by $length \times width \times height$ and the surface area is given by $2(l \times w + l \times h + w \times h)$.
- Write a function `DigitToInt` that takes a character digit between `#"0"` and `#"9"` and returns the corresponding integer from 0 to 9.

2.6.0.2. Successfully create and execute the following functions using If-Then or Case conditionals:

- Write a function that returns `true` if its input is greater than 0 and less than 5 and `false` otherwise.
- Write the function `isEven` that returns true if its input is an even integer.
- Write the function `implies` that takes two booleans LHS (left-hand side) and RHS (right-hand side) and returns the appropriate truth value according to the following truth table:

| LHS | RHS | LHS $\rightarrow$ RHS |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

- Write the function `clampValue` that returns zero if its input is less than zero, returns ten if its input is greater than ten, and returns the input otherwise.

- Write a function `IntToDigit` that takes an integer 0-9 and returns the corresponding character `#"0"` to `#"9"`.
- Write a function `fn constantOperation : char -> int -> int -> int` that takes three parameters: one of the characters `#"+"`, `#"-"`, `#"*"`, or `#"/"`, and integers LHS (left-hand side) and RHS (right-hand side), performs the operation `LHS op RHS` and returns the value.
- Idiomatic SML prefers the clausal form from "Functions, Case Statements, and a New Function Syntax." If you have not already, use the clausal form to re-write the functions `implies` and `clampValue`.

2.6.0.3. Use function currying and partial function application to complete the following exercises:

- Write a function `expandedClampValue` that takes three inputs: low, high, and inputValue. If the inputValue is less than low, then return low. If the inputValue is greater than high, then return high. Otherwise, return inputValue.
- If you have not already, re-write the function from above in curried form- that is, it should not take a tuple, and its type should be `fn : real -> real -> real -> real`
- Partially apply the values 0 and 10 to derive a second function that mimics `clampValue` from above. Test your function.
- Partially apply another set of low and high values to derive a third clamp function that behaves differently.

# Class 3

## 3.1 Recursion in ML

Recursion is a natural setting for writing many functional programs, as recursion inherently emphasizes statement execution. Operations such as traversing an array or linked list would be obviously non-recursive in other languages, but end up being done recursively in SML for this reason. Thankfully, SML is naturally structured to encourage recursion, and once you have made a mental shift into the functional paradigm you'll be thinking recursively too!

Idiomatic SML uses clausal functions to implement recursion. The basic pattern is:

```
fun myFunc base_case      = expression
  | myFunc recursive_case = myFunc( expression )
```

For example, the classic Fibonacci sequence is defined so that the $n^{th}$ element is the sum of the previous two elements. There are two base cases for 0 and 1:

$fib(0) = 0$

$fib(1) = 1$

$fib(n) = fib(n-1) + fib(n-2)$

Writing the fibonacci function in SML is almost the same:

Program `fib.sml`

```
1  fun fib 0 = 0
2    | fib 1 = 1
3    | fib n = fib(n-1) + fib(n-2);
4
5  fib(0);
6  fib(1);
7  fib(2);
8  fib(3);
```

```
 9  fib(4);
10  fib(5);
```

Which gives:

```
[opening fib.sml]
val fib = fn : int -> int
val it = 0 : int
val it = 1 : int
val it = 1 : int
val it = 2 : int
val it = 3 : int
val it = 5 : int
```

For another example, consider the famous Collatz Conjecture. We define the following function:

$$f(n) = \begin{cases} \frac{n}{2}, & \text{if n is even} \\ 3n+1, & \text{if n is odd} \end{cases} \tag{3.1}$$

The Collatz Conjecture says that if we repeatedly apply this function, then we eventually reach the number 1 for any starting value $n$. For example, if we start with $n = 6$ then we get the sequence $[6, 3, 10, 5, 16, 8, 4, 2, 1]$. We start at six, and since it is even we divide down to three, then we multiply and add to get 10, then divide down to 5, then multiply and add up to 16, etc. As far as we know the conjecture holds true for millions of inputs, but so far a conclusive proof has been elusive.

Computing the Collatz sequence is obviously recursive. Suppose we want to write a function `collatzLength` which, for an input $n$ will tell us how many steps it takes to get to the value 1. For example, `collatzLength( 6 )` would yield the value 9, from the sequence above. We can easily compute the Collatz sequence itself by translating the recursive definition. Getting the length is just a matter of counting how many recursive calls are made. We can recursively count backwards, considering 1 to be the first element of the sequence counted, and then incrementing our count each time the recursive call unwinds.

Program `collatzLength.sml`

```
1  fun isEven n =
2    if n mod 2 = 0
3    then true
4    else false;
```

```
5
6  fun collatzLength 1 = 1
7    | collatzLength n = if isEven n
8                        then 1 + collatz (n div 2)
9                        else 1 + collatz (3*n + 1);
10
11 collatzLength 6;
```

While this code computes the Collatz sequence length, it doesn't provide the sequence itself or print it out. We could think about hacking a print statement in there, and you're welcome to try, but we'll see a better approach shortly. When run, it does indeed produce the length 9. For now, let's hold off on that and introduce Lists instead.

## 3.2 Tuples and Lists

Tuples are ordered collections, where each element of the tuple does not have to have the same type.

Program `tuples.sml`

```
1 val coords = (1.5, 4.25);
2 val personAge = ("David", 35);
3 val tupleOfTuples = ( (1.5, 4.25), ("David", 35) );
```

```
- use "tuples.sml";
[opening tuples.sml]
val coords = (1.5,4.25) : real * real
val personAge = ("David",35) : string * int
val tupleOfTuples = ((1.5,4.25),("David",35)) : (real * real) * (string * int)
```

Lists are a collection of data. Each element of a list must have the same type, unlike tuples. Functional programming relies on using lists quite a lot. Lists can easily be defined recursively (inductively), making them a natural setting for functional reasoning.

Program `lists.sml`

```
1 (* Declaring lists explicitly *)
2 val emptyList = [];
3 val animals = [ "cat", "dog", "mouse" ];
4 val favoriteIntegers = [ 0, 1, 42, 2048, 9000 ];
5
6 (* Accessing lists *)
```

```
 7  (* Returns the head of the list *)
 8  val a = hd animals;
 9  (* Returns the tail of the list *)
10  val b = tl animals;
11
12  (* Modifying lists *)
13  (* The :: operator appends to the front of a list *)
14  val newFaves = 1337 :: favoriteIntegers;
15  (* We can also concatenate lists with @, or append to the back *)
16  val powers = [1, 2, 3] @ [10, 20, 30] @ [100, 200, 300];
```

```
- use "lists.sml";
[opening lists.sml]
val emptyList = [] : 'a list
val animals = ["cat","dog","mouse"] : string list
val favoriteIntegers = [0,1,42,2048,9000] : int list
val a = "cat" : string
val b = ["dog","mouse"] : string list
val newFaves = [1337,0,1,42,2048,9000] : int list
val powers = [1,2,3,10,20,30,100,200,300] : int list
```

## 3.3 Inductive Defintion of Lists in SML

Lists are especially useful for writing recursive code, and the `::` operator is crafted so we can operate on lists recursively. Note the definition of `::` requires that the left hand side operator be a list element, while the right hand side is a list. The statement `5 :: 10` will give an error, because `10` by itself is a number, not the list containing the number ten. The statement `5 :: [10]` appends the value five to the list containing the element ten.

We can define lists inductively using this. The empty list is given by `nil`, or two square brackets with nothing inside `[]`. Given the discussion of the `::` operator above, a list containing a single element can be created by appending with `nil`:

```
[ 5 ] = 5 ::  nil
```

And the two-element list `[5 10]` can be given by:

```
[ 5 10 ] = 5 ::  ( 10 ::  nil )
```

Thinking inductively, a longer list can be defined similarly:

```
[ 5, 10, 15, 20, 25 ] = 5 ::  (10 ::  (15 ::  (20 ::  (25 ::  nil ))))
```

And additionally, the parentheses are actually unecessary because the `::` operator is *right-associative* (most of our other operators tend to be *left-associative*) so we can shorten the above definition to:

```
[ 5, 10, 15, 20, 25 ] = 5 ::  10 ::  15 ::  20 ::  25 ::  nil
```

## 3.4 List Recursion in ML

Using the definition of `::` given above, with SML's case-statement pattern matching allows us to write recursive functions over lists. This is especially useful for "list processing." In other languages where you would write a for-loop to read every element of a list in turn, or apply an operation to every element of a list, in idiomatic SML you would use recursion. For example, if we wanted to sum all of the elements of a list:

Program `sumList.sml`

```
1  fun sumList nil = 0
2    | sumList (x::xs) = x + (sumList xs);
3
4  sumList [1, 2, 3, 4, 5];
```

This heavily relies on SML's pattern matching engine, and recall that `nil` is the empty list. The pattern `x::xs` matches any list with one or more elements, so when `sumList( [1,2,3,4,5] )` is called, this is translated into `1 + sumList( [2,3,4,5] )`. Recall also that `5::[]` is the same as the list containing `[5]`, so it matches as well.

If instead of reading the list we wanted to modify the list, we can do that with recursion as well. Suppose we want to add ten to every element of a list:

Program `listAddTen.sml`

```
1  fun listAddTen nil = nil
2    | listAddTen (x::xs) = (x+10)::listAddTen(xs);
3
4  listAddTen [3,5,7];
```

Note that we are actually consuming the old list and creating an entirely new list. The new list is created incrementally on the right hand side of the assignment statement. Pay attention also to the types being used. Here the base case is `nil` because the recursive case appends to a list, so the base case has no effect. If we tried to append zero, we'd

first have a syntax problem, but even so we don't want to append the element zero to our list in any case. If we expand the recursive call out we'd get the following:

```
(3 + 10)::(5 + 10)::(7 + 10)::nil
```

Which gives the desired result.

While we can write code in the above fashion, idiomatic SML uses the `map` function when modifying lists. Rather than writing a recursive function and hoping the result is clear code, the `map` approach explicitly applies a specified function to every element in a list. That is, if $f$ is a function and *list* is a List with $n$ elements, then:

$$map \;\; f \;\; list \;\; = \;\; [f(list_1), f(list_2), ..., f(list_n)]$$

We can re-write the addTen example above using map:

Program `mapAddTen.sml`

```
1  fun addTen x = x + 10;
2
3  val myList = [4, 6, 8];
4
5  val result = map addTen myList;
```

The resulting program is significantly easier to write and understand over using list-recursion.

We have seen recursive examples of reading through a list and modifying a list, lastly we will see an example of building a list recursively. Remember our function `collatzLength` from the previous section, and how we were unable to see the sequence generated by the Collatz function. We fix this in our code below. The code is the same, except we take care to add each recursive result to a list, and the base case must start the list by adding to `nil`. Note also that our function no longer returns the length of the Collatz sequence, but instead we can use the `length` function to get that value on demand, or we could even define a new `collatzLength` function using the code below and `length` together.

Program `collatz.sml`

```
1  fun isEven n =
2    if n mod 2 = 0
3    then true
4    else false;
5
6  fun collatz 1 = 1::nil
7    | collatz n = if isEven n
8                        then n::collatz (n div 2)
```

```
 9                        else n::collatz (3*n + 1);
10
11  val c = collatz 6;
12  length c;
```

## 3.5 Class 3 Exercises

3.5.0.1. Implement the following functions using recursion:

- The function `factorial` which takes an input `x` and computes the product $1 \times 2 \times 3 \times ... \times x$.
- The function `sumByFives` which takes an input `x` and computes the sum $x + (x - 5) + (x - 10) + (x - 15) + ...$ for all values greater than zero. Hint: Use an if-then statement instead of pattern matching.
- The function `sumBetween` which takes two inputs `x` and `y`, and returns the sum $x + (x + 1) + (x + 2) + ... + y$. Hint: Use an if-then statement instead of pattern matching.

3.5.0.2. Implement the following functions using list recursion:

- The function `makeInts` which takes as input an integer $x$ and returns the list $[x, ..., 3, 2, 1]$.
- The function `myLength` which takes as input a list and returns its length.
- The function `reverse` which takes as input a list and reverses it.
- The function `sumPositive` which takes a list of integers and sums up only the positive elements.

3.5.0.3. Implmement the following using the map function. You may define the argument to map as a regular named function, or use an anonymous function:

- Using the function `makeInts` from above, create a sequence of even integers $[x, ..., 8, 6, 4, 2]$
- Using the function `makeInts` from above, create a sequence of odd integers $[x, ..., 7, 5, 3, 1]$
- Using the list from the previous exercise, create the inverse sequence, which is $[\frac{1}{1}, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, ...]$
- Create an alternating list from the previous sequence: $[\frac{1}{1}, \frac{-1}{3}, \frac{1}{5}, \frac{-1}{7}, ...]$
- Sum the alternating inverse sequence, and multiply by four, which should yield an approximation of Pi. The approximation should become more accurate with more terms

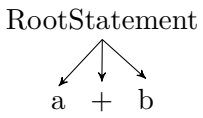
# Class 4

## 4.1 Tree Data Structures

Frequently we will use a tree data structure to pass data between modules. Trees end up being really nice data structures for capturing syntax and semantics. Suppose we have the statement:

a + b

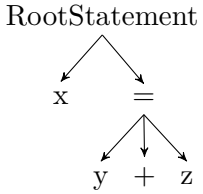
This can be naturally represented as a small tree:




A slightly more complicated statement:

x = y + z

Can also be represented as a tree with nested subtrees:




And note that this tree structure gives us some nice properties that we may end up using later. See Appel pp 7-9 and Figure 1.4 in particular for an expanded discussion of tree representation.

## 4.2 Datatypes in SML

Before we get to building trees, let's first discuss the `datatype` keyword. It allows the programmer to create a new type that can be recognized by the SML type system. For example,

```
datatype coins = penny | nickle | dime | quarter
```

Each element of a type is automatically provided the equality and not-equality operators, so `penny = penny` and `nickle <> dime`, but not `penny = quarter`.

Moreover, this these elements can then be used by the pattern matching engine:

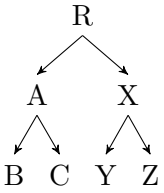
Program `coins.sml`

```
1  datatype coins = penny | nickle | dime | quarter;
2
3  fun coinToValue penny   = 1
4    | coinToValue nickle  = 5
5    | coinToValue dime    = 10
6    | coinToValue quarter = 25;
7
8  fun valueToCoins 0 = nil
9    | valueToCoins x =
10       if x >= 25 then
11         quarter::valueToCoins(x-25)
12       else if x >= 10 then
13         dime::valueToCoins(x-10)
14       else if x >= 5 then
15         nickle::valueToCoins(x-5)
16       else
17         penny::valueToCoins(x-1);
```

## 4.3 Trees in SML

SML does not contain a built-in tree data structure that we can use, however, we can define one readily using the `datatype` command from above. Let's first consider an unrestricted binary tree. We will say that each node of the tree contains up to two elements.

Importantly, SML allows recursive `datatype` declarations. This is critical, because trees have a natural recursive definition. Consider the following tree:




Here, the subtree ABC is an otherwise valid tree on its own, and the same can be said for the subtree XYZ. If we adopt the syntax `tree( node, left-subtree, right-subtree)` then we can recursively define the entire tree:

```
tree( R,
    tree(A, B, C),
    tree(X, Y, Z)
    )
```

Thus, when it comes to defining a tree in SML, we give a recursive definition of a tree of trees:

```
datatype tree = Empty
              | Leaf of char
              | Node of char * tree * tree
```

According to the above, a tree may be the `Empty` value, which we will consider analogous to the empty list, or the tree is a tuple named `Node` containing a char and two subtrees. A subtree may also be a `Leaf` containing a single char value. By convention we will assume the second argument is the left subtree and the third argument is the right subtree. Thus, we can formally define tree from above in SML as:

```
val myTree = Node(#"R",
                  Node(#"A",
                       Leaf(#"B"),
                       Leaf(#"C")
                      ),
                  Node(#"X",
                       Leaf(#"Y"),
                       Leaf(#"Z")
                      )
                 )

val sameTree = Node(#"R",
```

```
                    Node(#"A", Leaf(#"B"), Leaf(#"C")),
                    Node(#"X", Leaf(#"X"), Leaf(#"Y")));
```

Taking our definition one step further, there's nothing inherent about `char` that makes it part of a tree. Our tree datatype should reflect the structure of the tree, but doesn't need to reflect the contents of the tree. This is one place where we can use a *arbitrary type*.

```
datatype 'a tree = Empty
                 | Leaf of 'a
                 | Node of 'a * 'a tree * 'a tree;
```

Note that our datatype definition has changed slightly. Before we could refer to the `tree` type, which was unambigously referring to a tree containing `char` values. We can use the above definition to create a tree containing any arbitrary type, such as a tree of `int`, `char`, or `string` types. However, the arbitrary type now becomes part of the tree's declaration. We no longer have a `tree` type, we have a `'a tree` type, which specializes to `int tree`, `char tree`, etc. Both of the following declarations work equally without extra syntax now, as the type system figures things out on its own:

```
val sameTree = Node(#"R",
                    Node(#"A", Leaf(#"B"), Leaf(#"C")),
                    Node(#"X", Leaf(#"X"), Leaf(#"Y")));

val sameTree = Node(5,
                    Node(1, Leaf(2), Leaf(3)),
                    Node(7, Leaf(8), Leaf(9)));
```

## 4.4 Tree Access

Right now our tree structure is totally opaque- we can unpackage our values from the tree using the pattern matching engine.

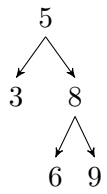
```
fun access (Leaf(x)) = x
  | access (Node(x, left, right )) = x;

fun left (Node(x,left,right)) = left;
fun right (Node(x,left,right)) = right;
```

More likely, we might also want to search the entire tree to see if a given key exists within the tree. Of course, we can use the recursive structure of the tree to make this easier. We search at the current node, and if we match we return true, otherwise we search the left and right subtrees.

```
fun exists x Empty = false
  | exists x (Leaf(y)) =
     if x = y then
       true
     else
       false
  | exists x (Node (y, left, right)) =
     if x = y then
       true
     else
       (exists x left) orelse (exists x right );
```

## 4.5 Binary Search Tree Insert

Sometimes trees can be given in finished form as we did previously, but it is typically more common to build a tree dynamically with an `insert` function. Suppose we want to build a *binary search tree*, which has the property that every node to the left of a node is less than the current node value, and everything to the right of a node is greater than. This requires us to traverse the tree.




For example, in the above tree, inserting 1 would require us to make a left subtree of 3, while inserting while inserting 4 would require a right subtree of 3. Similar to the `exists` function, we handle each type of the `tree` datatype, but in this case we return a new tree of appropriate structure.

As before, there are three cases to consider. If the tree argument is the empty tree, then we replace it with a leaf containing the new value. If the tree argument is a leaf, we create a new node with left and right subtrees. However, one of those will be Empty, and the other will contain the new value, depending on whether the new value is greater or less than the node. If the tree argument is a node itself, then we must insert into

the left or right subtree, depending the new value. When doing so, we don't just want to recursively call `insert`, or else the returned value would be only the final location traversed by `insert`.

```
fun insert x Empty = Leaf(x)
  | insert x (Leaf(y)) =
    if x < y then
      Node(y, Leaf(x), Empty )
    else
      Node(y, Empty, Leaf(x) )
  | insert x (Node(y, left, right)) =
    if x < y then
      Node(y, (insert x left), right)
    else
      Node(y, left, (insert x right));
```

## 4.6 Class 4 Exercises

4.6.0.1. Use the SML `datatype` keyword to create a functional color system:

- Define a `color` datatype that contains the colors: red, green, blue, yellow, cyan, magenta, black and white.
- Define a new addition operator that combines the `color` types in accordance with the theory of additive color. Under this color system, red, green, and blue are primary colors. Black is the absence of color, and white is the combination of all three primary colors. Red and Green combine to make Yellow, Red and Blue combine to make Magenta, and Green and Blue combine to make Cyan. Thus, your color system would include the following productions, and others as well:

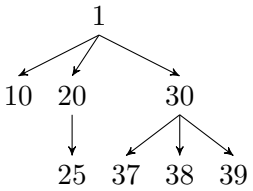
$$Black + Red \mapsto Red$$
$$Red + Green \mapsto Yellow$$
$$Yellow + Blue \mapsto White$$
$$White + Magenta \mapsto White$$

  The addition operator is an *infix* function, meaning that it takes arguments from its left and right sides, rather just from the right side. (Normally most functions are *prefix* functions, and we typically don't use *postfix* functions in programming.) You can define a new function on the addition operator using the `op` keyword. For example:

```
fun op + (Red, Green) = Yellow
```

4.6.0.2. Use the SML `datatype` keyword to create a functional 3-tree datatype.

- Define a 3-tree structure- that is, a tree data structure where each node has an internal value, plus up to three outgoing branches. These branches may terminate in other nodes, leaves, or the Empty tree.
- Create the following tree using your data structure:




- Define a function `leftmost` that takes a 3-tree and returns its leftmost value- the value gotten by traversing leftward at each Node until you reach a leaf.
- Define a function `exists` that takes a 3-tree and a value, and returns True if the value exists in the tree, and false otherwise.

# Class 5

## 5.1 Compiler Data Structures

Recall that a compiler is fundamentally a language translator. Each stage of a compiler may be considered an individual language translation. The sum of these multiple translations is the overal transformation from source code to executable code.

For compiler tasks, a key question is how the program is represented. Different representations have different advantages. Source code in a text file is convenient for programmers to read and modify, but would be awkward to do compilation and processing with. (Such a thing exists though- *source-to-source* compilers take source code in one language and transform it to source code in another language, however these may also use internal data structures that are not raw source code.) Conversely, a data structures such as a trees or lists might be harder for a programmer to use directly, but easier for a program to process. In this section we explore an example usage of the tree data structure in representing a program.

## 5.2 A Tree Grammar

Rather than trying to represent a program in source code, we will use a tree. To be precise, we first define a specific *tree grammar*- a set of rules about what our tree is allowed to contain and how we can put trees together. This tree represents our program, so it is also the grammar that defines the rules of a simple programming language.

The tree grammar is given in table 1.3 in chapter 1. We will talk about grammars more later, but for now we say that a grammar rule *produces* a new representation when it is applied. For example, if we start with a single statement: `stm` then we can produce a new program by applying one of the top-left three rules in the table. The first produces `stm` $\mapsto$ `stm; stm`. Note that this production is *allowed* but it is not *required*. We could instead apply the second production and get `stm` $\mapsto$ `id := Exp`, then applying the fourth and fifth productions we could transform `id := Exp` $\mapsto$ `x := 5` Here there are no more tokens that produce anything, so our final program is just "x = 5".

The process of repeatedly applying productions until there are no "unresolved" tokens

left enumerates all possible (infinitely many) valid programs under the grammar. A key job of the compiler is to take a source code file as input, and then try to understand the sequence of productions that could yield that file. Programs that cannot be produced under the grammar would yield a compiler error.

## 5.3 Straight-Line Program Representation in SML

The SML code given in Program 1.5 is derived from Grammar 1.3 and represents a set of datatypes that are sufficient to describe our straight-line language. This looks different from the tree definition we studied in class together, but note that it does indeed define a tree structure when interpreted as such. (Our own definitions of a 2-tree didn't strictly define a "tree" either- only the pictures we drew on the board interpreting those definitions looked like trees!)

Note that this definition is a recursive definition, as is required to define a tree this way. Each `compoundStm` statement yields two additional statements, while `OpExp` and `EseqExp` both yield additional expressions.

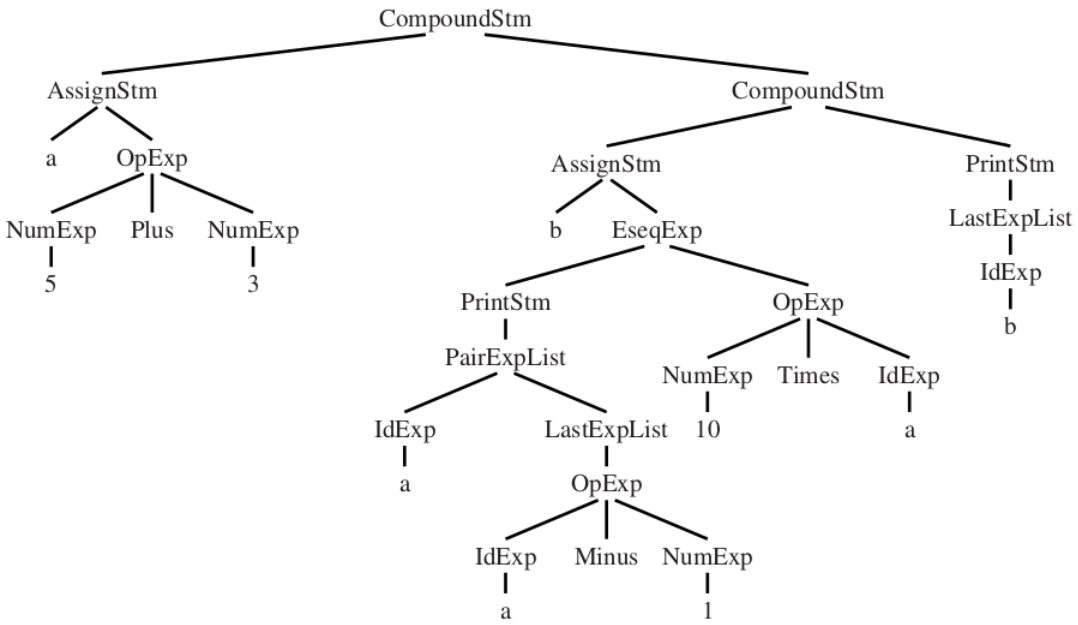
```
type id = string

datatype binop = Plus | Minus | Times | Div

datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list

     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
```

**PROGRAM 1.5.** Representation of straight-line programs.

Figure 5.1: From Appell Pg. 9

## 5.4 Program 1: A Straight-Line Interpreter

Our first programming project is to write the Straight-Line Interpreter described starting on page 10 of Appell. This is similar to an interpreter such as the Python or SML




```
a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
```

Figure 5.2: From Appell Pg. 9

interepreter- it takes a series of expressions and evaluates them for their effects. Unlike those interpreters your program does not operate on a character stream, instead you are given the program tree directly using the ML definitions given in Program 1.5 on page 9. `prog` given on page 10 is an example input to your program. If we invoke the high level function `interp` such as `interp prog`, then the output would be:

```
8  7
80
```

This sounds like a lot, and it is. As is the case in any programming challenge, we can break this large task down into smaller tasks to get ourselves started. You are asked to write, at a minimum, three functions:

- A high level driver function, `interp stm` $\mapsto$ `unit`
- A statement interpreter, `interpStm stm * table` $\mapsto$ `table`
- An expression interpreter, `interpExp exp * table` $\mapsto$ `int*table`

However, I found myself writing several other support functions as well. These supports may be easier to write and are probably the place to get started!

- A function that updates a symbol table, `update ( table, ID, value)` $\mapsto$ `table`
- A symbol table lookup function, `lookup (table, ID)` $\mapsto$ `int`
- A function to handle OpExp expressions, `doBinaryOp (int, binop, int)` $\mapsto$ `int`

Some general hints:

- Your interpreter must evaluate statements from left to right. If we have the `CompoundStm` that is "x = 3; x = x*4" then we expect x to be defined first and the total output to be 12.
- You will need to implement a symbol table- this is a list of all current symbols (IDs) in the program with their current value. For example, at the end of `prog` the symbol table would contain the mappings $\{a \mapsto 8, b \mapsto 80\}$
- You will want to impelment a high level function `interp`, and (at least) two helper functions `interpStm` and `interpExp`, which interpret a statement and expression respectively.
- The symbol table is the "state" of the interpreted program, and new statements/expressions can only be interpreted alongside that table.
- High level statements can modify the symbol table, so `interpStm` should have type `stm` $\times$ `table` $\rightarrow$ `table`.
- Expressions can modify the symbol table, but can also return a value immediately. This is necessary for expressions containing `EseqExp` such as "y := (x := 5, x + 10)". As a result, `interpExp` should have type `exp` $\times$ `table` $\rightarrow$ `int` $\times$ `table`.

Note that you can/should use the declarations from page 9 of the book verbatim. These will get you started, and essentially define the work you have to do. The `interpStm` function needs to handle all three different types of statements. The `interpExp` needs to handle all four types of expressions. Those definitions are, repeated here for convenience:

```
(* Definitions from Appell *)
type id = string

datatype binop = Plus | Minus | Times | Div
```

```
datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list

     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
val prog =
 CompoundStm(AssignStm("a",OpExp(NumExp 5, Plus, NumExp 3)),
  CompoundStm(AssignStm("b",
      EseqExp(PrintStm[IdExp"a",OpExp(IdExp"a", Minus,NumExp 1)],
           OpExp(NumExp 10, Times, IdExp"a"))),
   PrintStm[IdExp "b"]))
```

Some required/useful SML syntax we haven't covered yet in these notes:

### 5.4.1 Mutually Recursive Functions and Datatypes

SML supports functions that are *mutually recursive* using the `and` keyword. Mutual recursion happens when two (or more) functions rely on each other, creating a cyclic dependency. For example, the following pair of functions work together to create a series of alternating positive and negative values- calling `alt 5` results in the list `[5, ~4, 3, ~2, 1]`:

```
fun alt  0 = []
  | alt  x = x::alt2(x-1)
and alt2 0 = []
  | alt2 x = ~x::alt(x-1);
```

These functions are mutually recursive, because `alt` calls `alt2`, and `alt2` calls `alt`. **Note the use of the `and` keyword, and notice that we only use the `fun` keyword once.** Without the `and` keyword, SML would complain "unbound variable or constructor: alt2"- in other words, it doesn't know what `alt2` is. In C this would be accomplished with a *forward declaration*- promising the compiler that a function with a given name exists, but you aren't defining it yet. In Python this is solved by *late binding* functions to names only at runtime when they are called. In JavaScript this is solved by automatically *hoisting* function definitions to the top of their scope.

We can also define mutually recursive datatypes using the same `and` keyword. Like with functions, this happens when two datatypes depend on knowledge of each other, creating a circular dependency. This can occur when we have two concepts that are defined in terms of each other.

In our Straight Line Interpreter, our definitions of the `stm` and `exp` datatypes (provided by Appell) are mutually recursive. This is because `AssignStm` and `PrintStm` are statements that include expressions, and because `EseqExp` is an expression that includes a statement.

Similarly, the functions `interpStm` and `interpExp` are mutually recursive, because some statements require interpreting expressions, and some expressions require interpreting statements. They should be declared as such:

```
fun interpStm pattern1 = ...
  | interpStm pattern2 = ...
  | interpStm pattern3 = ...
and interpExp pattern1 = ...
  | interpExp pattern2 = ...
  | interpExp pattern3 = ...
```

### 5.4.2 Temporary Variables with `Let/In`

Temporary variables allow us to write more readable code, while making some things explicit:

- Deconstruction of tuples into components
- The order that subexpressions are evaluated
- How many times a subexpression is computed

For example, the following code demonstrates these uses of temporary variables in computing `a`, `b`, and `c`:

Program `let_in.sml`

```
1  val t = (5, 10, 15);
2
3  fun twiddle (input : int*int*int)  =
4    let
5      val (x, y, z) = input
```

```
 6      val a = x + y
 7      val b = a + z
 8      val c = a + b
 9    in
10      (a, b, c)
11    end;
12
13  twiddle t;
```

This will come in handy when writing multiple clauses of `interpExp` and `interpStm`, at least. For example, the `OpExp` expression contains two expressions. Each expression is capable of modifying the global symbol table, so some code may execute differently depending on whether we evaluate the left expression first or the right expression first. We would like to specify the order of execution, as well as which version of the symbol table goes with the evaluation of the left and right expressions. Thus, for this clause of `interpExp` we could write:

```
fun interpExp (OpExp( left, operator, right ), table) =
  let
    (* Evaluate the left expression *)
    (* Evaluate the right expression *)
  in
    (* Compute the operation result and return (result, newTable) *)
  end
```

However, this is not the only place where it might make sense to use a `let` block in our Straight Line Interpreter.

### 5.4.3 Printing values

SML includes a basic facility for printing strings- the `print` function. You will need to use this function to properly interpret the `PrintStm` statement (i.e. produce output). This function operates only on strings, but the basic types can be easily converted to strings, and strings are easily sliced, concatenated, and otherwise modified. For example:

```
     print "42
n";
```

The above prints the string “42” followed by a newline character. If we had an integer, we would have to convert it to a string before printing it:

```
val myInt = 2358;
print ( Int.toString(myInt) ^ "\n" );
```

The above prints the number 2358 followed by a newline. The ^ character is the string concatenation operator. The parentheses are required, because function application is a high-priority operator.

However, one issue arises in using `print`- as a function, it must return something, which is the unit: `{}`. The unit is the closest that SML gets to not returning anything from a function, but it is still a real returned value. This can cause confusion, for example, the following code does not work:

Program `bad_add_print.sml`

```
1 fun add_and_print x y =
2 let
3   val sum = x + y
4 in
5   print (Int.toString(sum)^"\n")
6   sum
7 end
```

That function fails to interpret with the following error:

```
bad_add_print.sml:5.3-6.6 Error: operator is not a function [tycon mismatch]
  operator: unit
  in expression:
    (print (Int.toString sum ^ "\n")) sum
```

What happened? Remember that whitespace is largely ignored in SML. The `print` function returns the unit, which is the applied to the next token on the next line. However, "applying the unit" to anything doesn't make sense- it is not a function, hence the error we get. The solution is to just put a semicolon at the end of the print statement on line 5. This separates the two expressions so they don't affect each other, which in this case has the effect of suppressing and ignoring the return value of `print`, which is what we want. We can use this to insert a print statement anywhere we want, perhaps for debugging purposes. For exmaple, the expression ( print "here
n"; 25 ) causes "here" to be printed to the terminal and executes the expression "25," returning the integer 25. This is necessary, when printing a list, for example:

Program `print_list.sml`

```
1 fun print_list nil = print("\n")
2   | print_list (x::xs) =
```

```
3       (print( Int.toString(x) ^ " " );
4       print_list(xs))
```

Notice the outer parentheses on lines 3 and 4- they are necessary. Without these, we would have the semicolon on line 3, and the result would be that the interpreter considers the fourth line to be separate from the first three. In effect, the interpreter would forget what `xs` was and would give an error about an unbound variable.

## 5.5 Class 5 Exercises

5.5.0.1. Answer the following questions about the straight-line grammar in table 1.3:

- Starting with just `stm`, pick and apply five production rules
- Give the sequence of productions that yields the program, "x = 5; y = 10; print (x + y)"
- Give the sequence of productions that yields the program, $x = \frac{5+10}{20}$
- Give an example of a program that cannot be produced by the grammar

5.5.0.2. Construct the following program trees using ML data constructors, following the example shown on page 10:

- x = 5; y = 10; print (x + y)
- $x = \frac{5+10}{20}$

5.5.0.3. Answer the following questions from Appell, pg 12:

- You are provided the basic implementation of a binary search tree in exercise 1.1. Read/run this code and make sure you understand it.
- Complete exercises a, b, and c. For part b, interpret this as the generalization of the tree to data types such as `int` and `real`.

# Class 6

In this class we discuss the differences between lexical, syntactic, and semantic analysis, and talk about regular expressions as a form of *lexing* - performing lexical analysis.

## 6.1 Lexical, Syntactic, and Semantic Analysis

Recall the archetypical three-stage compiler. The front-end analyzes a specifica programming language input and produces a language-independent intermediate representation. The middle stage performs optimizations on that intermediate representation. The back-end produces an executable instruction stream that is specific to a single output machine-architecture. When we talk about language analysis— lexing, syntax analysis, and semantic analysis— we're talking about that front-end component. The same kinds of analysis techniques can be used for multiple languages, but a single compiler front-end is built to analyze a specific language.

Appell defines these three sections thus:

| | |
|---|---|
| **Lexical:** | Breaking the input into individual words or tokens |
| **Syntax:** | Parsing the structure of a program |
| **Semantics:** | Calculating the program's meaning |

## 6.2 Lexical Analysis

Lexical analysis is performed by a *lexer*, also called a *tokenizer* or *scanner*.

Lexical analysis requires (1) deciding what characters/tokens of an input are important, and what is not, and (2) a strategy for generating a tokenized input stream. For an example of item one, we have seen how whitespace is almost totally ignored in SML. Those characters have no syntactic or semantic meaning, and a lexer only cares about them insofar as they separate other, more meaningful tokens. This is not always the case: Python uses indentation— spaces or tab characters at the start of a line— to determine program scope. Thus, spaces and tab characters in Python have special meaning when they are placed at the start of a line, and the lexer must account for this. Another

example- in many languages (such as MATLAB) the end-of-line marker (typically an ASCII carriage return and/or line feed character) has the effect of separating statements in the language. However, the programmer may want to write long statements without having them awkwardly wrap around the screen, so one language feature is to use an ellipsis (...) to suppress that end-of-line marker.

For example, the following code might produce a tokenized sequence:

```
x = y + 10
```

```
id(x) EQUALS id(y) op(+) num(10) ENDLINE
```

For the second item, recognizing the content of tokens is not always easy. Suppose a language has a keyword LOOP. This typically does not preclude identifiers with names such as LOOK or LOOT.

Even recognizing what is a token may not be obvious. The following example is from Ron Cytron at Washington University in St. Louis. There are programming languages where whitespace is optional- as is the case in FORTRAN 66. In such languages, if you consume an input file character-by-character, you don't necessarily know what you are looking at until you find an apparently complete and sensible expression. A for-loop is specified in F66 with the keyword DO, followed by a line return number, then an index followed by a range. For example, the following statements are virtually identical when the whitespace is removed:

`DO 10 I = 1, 5` or without whitespace `DO10I=1,5`

and

`DO 10 I = 1.5` or without whitespace `DO10I=1.5`

The first statement is the beginning of a for-loop, while the second statement assigns the value 1.5 to the variable `DO10I`. They are only distinguished between the choice of colon or period between 1 and 5.

## 6.3 Regular Expressions (RegEx)

We want a more formal method for specifying the acceptable content of programs, and a more formal method for constructing lexers that recognize that content appropriately.

Formally, we will say that a *language* is a (typically infinite) set of strings. A *string* is a finite sequence of symbols. A *symbol* is taken from a finite *alphabet*. Mathematically, we

say that an alphabet is a *set* of symbols, while a language is a *set* of strings. A string is an *ordered sequence* of symbols.

These are expansive definitions. For example, we could say our alphabet is the set $\{a, b, c\}$, and our language could be the set of strings $\{aaa, bbb, ccc, abc\}$. We can construct strings that do exist in the language, like $abc$, but there are plenty of strings that are not in the language, like $aab$ or $cba$.

Practically, we want our languages to be things like, "the set of all valid C++ programs." The that is an infinite set, so we can't list them like we did above. Instead, we can describe a set of rules that gives rise to the same set. This is the purpose of *regular expressions* (or *regex*). We say that a regular expression is capable of *recognizing* or *generating* a set of strings. We can give a precise definition to "the set of all valid C++ programs" and similar sets by careful application of regular expressions.

As we continue, remember that the point of regular expressions is to recognize strings, which are sequences of characters. When we recognize a valid C++ program as being in the language, we are referring to a long sequence that is every character in the program.

## 6.4 RegEx Rules

Remember that regular expressions recognize languages, and a language is a set of strings. The following table is a brief description of the RegEx rules. Beware that this table is somewhat loose with the formal language. See Page 17 in Appell for more details.

| | |
|---|---|
| **Symbol:** | **a** recognizes the string $a$ |
| **Alternation:** | **a** \| **b** recognizes the set of strings $a$ or $b$ |
| **Concatenation:** | **a**·**b** recognizes the string $ab$, often the dot is omitted and written **ab** |
| **Epsilon:** | $\epsilon$ recognizes the empty string " " |
| **Repetition:** | **a**$^*$ recognizes zero or more repetitions of $a$, which is |
| **a.k.a Kleene closure** | the set { " ", $a, aa, aaa, aaaa$, etc. } |

Table 6.1: A summary of regex rules

There are a few other regex notation- see Appell pg. 18 for details:

There are two more rules we need to consider when writing regex:

**Longest Match:** A regex will always match the longest possible substring, starting at the start of input.

| | |
|---|---|
| **Class:** | The class [**abcd**] is equivalent to ( **a** \| **b** \| **c** \| **d** ), we also have the shorthand [**a-d**] which is equivalent to the above. |
| **a*** | Repetition zero or more times |
| $\mathbf{a}^+$ | Repetition one or more times, and is equivalent to ($\mathbf{a}\cdot\mathbf{a}^*$) |
| **a?** | Optional zero or one occurrences, equivalent to ($\epsilon$ \| **a**) |
| **.** | A period character stands for any other character except newline, note that some regex systems may include the newline |
| **Literal:** | A string in quotes matches itself literally, e.g. “int” or “char” |

Table 6.2: Additional regex notation

**Rule Priority:** If an input string could match under two different regex, the first regex written down is the one that matches. Thus, the order that regex are listed does matter.

For example, suppose we have the rules:

| | |
|---|---|
| Keyword: | (“int” \| “char” \| “if” \| “else” ) |
| Identifier: | [a-zA-Z][a-zA-Z0-9]* |

If we submit the input “intArray” to these rules, then by the Priority principle we would first match “int” with the Keyword rule, and then match “Array” with the Identifier rule. If these two rules were in reverse order, then Identifier would match the entire input “intArray” by the Longest Match principle. In fact, if Identifier was written before Keyword, then the rule Keyword would never match anything- why?

## 6.5 Class 6 Exercises

6.5.0.1. See Exercises 2.1 on Appell pg. 34

6.5.0.2. See Exercises 2.2 on Appell pg. 34

# Class 7

Given a Regular Expression, how do we build a program to implement that RegEx?

## 7.1 Deterministic Finite Automata

A finite automaton consists of a set of *states* which are connected by a set of directed *edges*, which lead from one state to another. Each edge is labeled with a specific *symbol* from the input *alphabet*. All finite automata have a *starting state* and one or more *final states*. In the case of a *deterministic* finite automaton, also called a DFA, no symbol has more than one outgoing edge from any given state.

In operation, a finite automaton consumes input strings. Beginning at the starting state, the automaton matches the next symbol to an edge, and follows that edge to the new state. (Note that edges are allowed to form loops- so following an edge may take you back to the current state.) This consumes that symbol in the string, and then the process is repeated until there are no more symbols left and no empty-string transitions that may be followed. If the finite automaton ends in one of the final states, then it is said to *accept* the string, and otherwise it *rejects*.

For example, in the C programming language there are four types of integer literals. From `https://en.cppreference.com/w/cpp/language/integer_literal`, an integer literal can be a:

- *decimal-literal* is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- *octal-literal* is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7)
- *hex-literal* is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)
- *binary-literal* is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)

We can encode these different integer literals as a set of RegEx:

- [1-9][0-9]*
- 0[0-7]*
- 0[xX][0-9a-fA-F]$^+$
- 0[bB][01]$^+$

These have an equivalent DFA encoding as such:

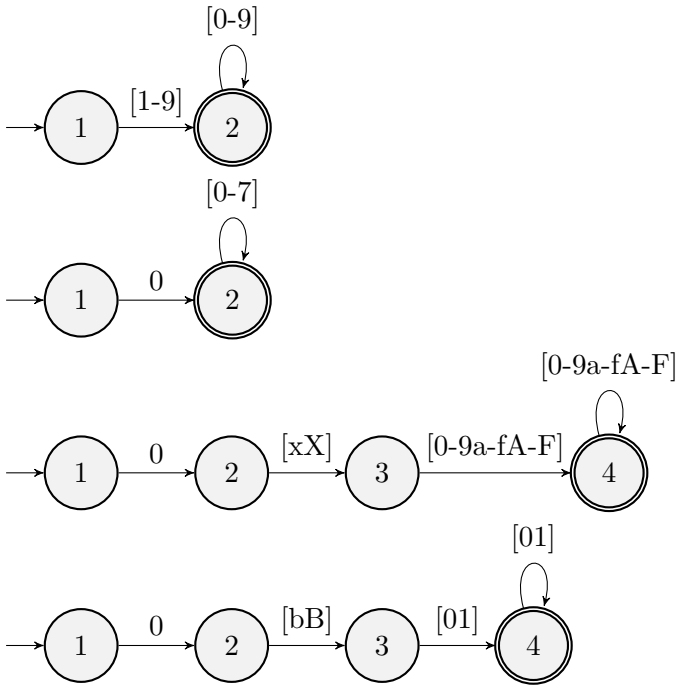



Figure 7.1: Individual DFAs for each C integer literal

Observe that the definitions of literal integers has been carefully selected to make sure that a lexer can unambiguously identify each type. For example, octal numbers are required to start with a zero, otherwise they could be indistinguishable from base-10 numbers. The string "123" is 123 in base-10, but is equivalent to 83 in base-8. Thus, it is possible to combine all four DFAs into a single DFA with four final states. By tracking which final state the machine ends in, the compiler can identify which type of integer literal is given.

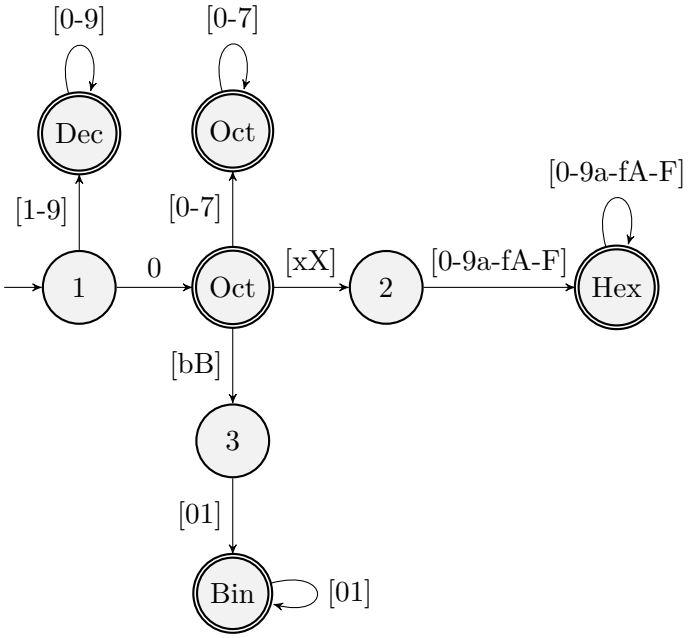



Figure 7.2: A combined DFA for C integer literals, with labeled end states

## 7.2 Rule Priority and Longest Match

We have two special rules for a collection of regex.

*Rule Priority* means that if a DFA final state can correspond to more than one regex rule, then it is labeled as the highest priority rule. That way, if two regex apply to the same final state, the highed listed regex is considered to match.

*Longest Match* means that a DFA should continue consuming input until the input is exhausted or there are no valid transitions. A DFA will never "stop early" as soon as it encounters a final state. In the C integer literal example above, the string "12345" should always match the decimal integer literal in its entirety. It should never match as "1" and "2345" or "12" and "345", etc.

However, it's possible for a DFA to move from a final state into a non-final state (though this does not happen in our example above, it does in the book's example). Thus, when processing under the longest-match rule we may "overshoot" the final match and continue processing input. To avoid this situation, we introduce two variables: the "last-final-state" and "position-at-last-final" variables. If the DFA processor finds itself stuck in a non-final state, then it can go back to the last time it occupied a final state, and this is the longest match.

## 7.3 Class 7 Exercises

# Class 8

Given a set of regular expressions, how do we combine them all into a single finite state machine?

## 8.1 Nondeterministic Finite Automata

A *nondeterministic* finite automata, or NFA, is defined the same as the DFAs previously, with two modifications. First, a single state may have multiple outgoing edges labeled with the same symbol. Second, states may have edges labeled with the empty string ($\epsilon$), which can be followed without consuming any input symbols. When processing on an NFA, you are allowed to take any valid move (consuming an input symbol if necessary). The NFA is said to accept a string if there is any possible set of moves that end in a final state. It is possible that there is more than one sequence of moves that will accept a string- and it may not always be obvious whether or not a given NFA can accept a given string!

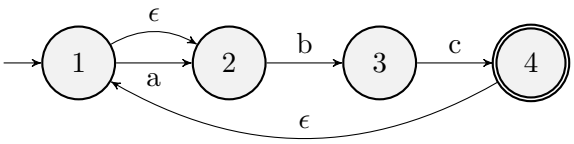



Figure 8.1: An example NFA with $\epsilon$ transitions

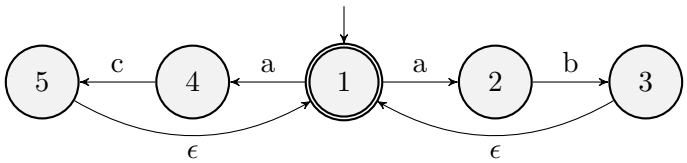



Figure 8.2: An example NFA with multiple same-symbol transitions

NFAs may be harder to process with, but they are useful because they are easier to construct from descriptions. DFAs are useful because they're easier to process with, but can be harder to construct (at least in an ad-hoc manner). For example, consider the C integer literal example from the previous class. Given the four machines that

accepted decimal literals, octal literals, hexadecimal literals, and binary literals, it took some effort to combine the four into a single machine capable of accepting any integer literal. Building that combination required analyzing each machine to see where they overlap, and figuring out where the branching points would be. However, the definition of an NFA allows easier synthesis, because acceptance with an NFA only requires that *some* sequence of transitions that consume the input. Combining those four machines is conceptually simple with an NFA- simply create a new initial state, and hook that new state to each machine with an $\epsilon$ transition.

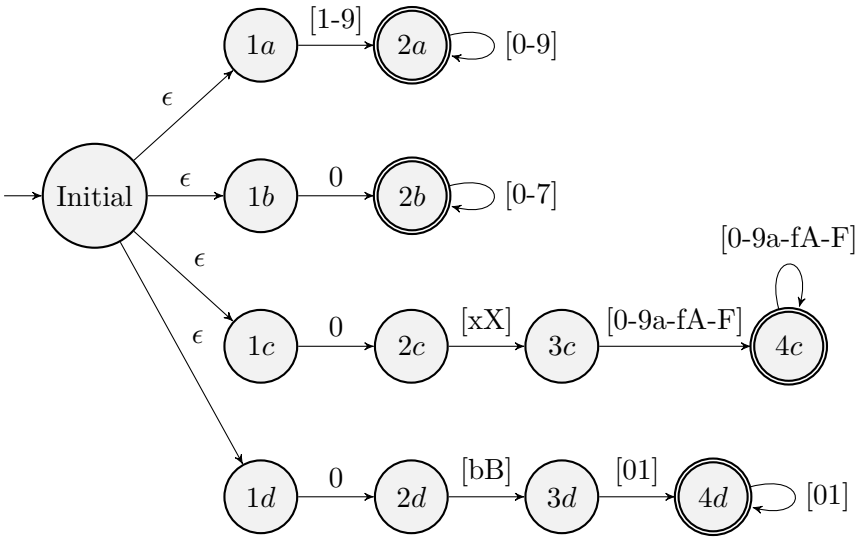



Figure 8.3: Individual DFA machines combined into a single NFA

## 8.2 Converting RegEx to NFAs

Recall that we had five basic regular expression rules: **Symbol**, **Alternation**, **Concatenation**, **Epsilon**, and **Repetition**. Our extended regex symbols like **?** and $^+$ were also defined in terms of those basic rules.

Each of these basic rules has an equivalent NFA construction. For example, consider the regex (**ab**)* - which accepts zero or more repetitions of *ab*. Using the recursive nature of RegEx, we can build an NFA that accepts this same language.

The machines in Figure 8.4 illustrate automata equivalents of the **Symbol** and **Concatenation** rules. Concatenation can be accomplished by "gluing" one automata to another, or more precisely, starting with the machine that accepts **a**, and then replacing the final state of **a** with the initial state of **b**. The result is the machine **ab**.

Similarly, converting the machine **ab** into **ab*** can be done mechanically, meaning that

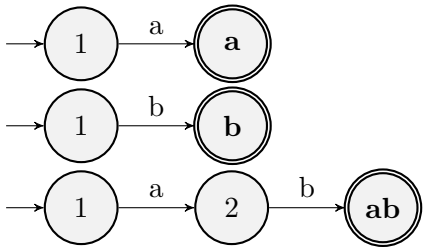



Figure 8.4: Example automata that accept the languages **a**, **b**, and **ab**

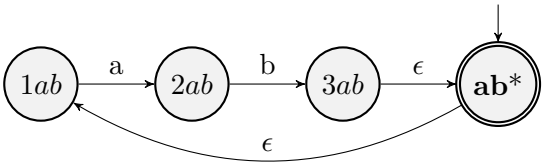



Figure 8.5: Example automata that accept the languages **a**, **b**, and **ab**

the RegEx **Repetition** rule can be recognized mechanically. Starting with the original **ab**, let the final state also be the initial state, and join the final state to the original initial state with an $\epsilon$ transition.

See Figure 2.6 or page 24 in Appell for more examples.

## 8.3 Converting an NFA to a DFA

While NFAs are perhaps easier to describe, construct, and combine in some senses, they are harder to process with. Deciding whether an NFA accepts a given string might require exploring many possible paths through the NFA. In contrast, processing a DFA involves no choices and does not require remembering what paths have been or might be tried, and they always consume exactly one character of input per transition. If you have a string of length $N$, the DFA will process and tell you whether it accepts or rejects in exactly $N$ steps.

Fortunately, we can get the best of both worlds. In the same way that RegEx can be algorithmically converted to NFAs, it is also the case that NFAs can be algorithmically converted to DFAs. A formal description of the technique is given on pages 25-28 of Appell, and I won't repeat it fully here. A brief motivating example, however:

The basic problem of computing on an NFA is that we might have to "guess" which of multiple transitions takes a given string to a final state. We can build a DFA that avoids this by building extra states that correspond to "guessing every transition simultane-

ously." If the NFA contains a state 1 that has an $\epsilon$-transition to another state 2, then we can build a combined state called $\{1, 2\}$ that combines the effect of simultaneously taking and not taking the $\epsilon$-transition, and this new state is called the $\epsilon$-closure of 1. The $\epsilon$-closure of a state combines every state reachable by an $\epsilon$-transition, even if multiple transitions are needed.

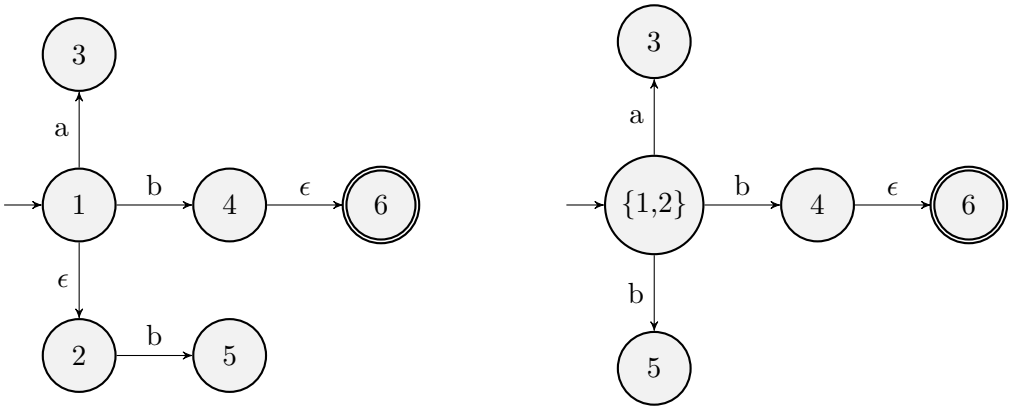



Figure 8.6: An example of computing the $\epsilon$-closure of state 1 in an NFA

However, merely removing $\epsilon$-transitions isn't enough to remove the nondeterminism from an NFA. We must also account for states where we have the option of following multiple edges- such as in the combined state $\{1, 2\}$ in Figure 8.6. Therefore we want to build yet another combined state that simulates taking both $b$ transitions simultaneously, which could be $\{4, 5\}$, but we notice that state 4 has another $\epsilon$-transition, so consuming the symbol $b$ at state $\{1, 2\}$ actually gets us to the trio of states $\{4, 5, 6\}$.

Thus, the NFA-to-DFA algorithm, or NFA simulation algorithm, can be described as thus: for every state $S$ and symbol $q$, create a new state that represents all of the states in the original NFA that are immediately reachable by following $q$ from $S$ (but without consuming additional input symbols). This is shown in Figure 8.7. More formally, for every state $S$ and symbol $q$, we compute the union of the $\epsilon$-closures of the states reachable from $S$ by consuming $q$.

## 8.4 Class 8 Exercises

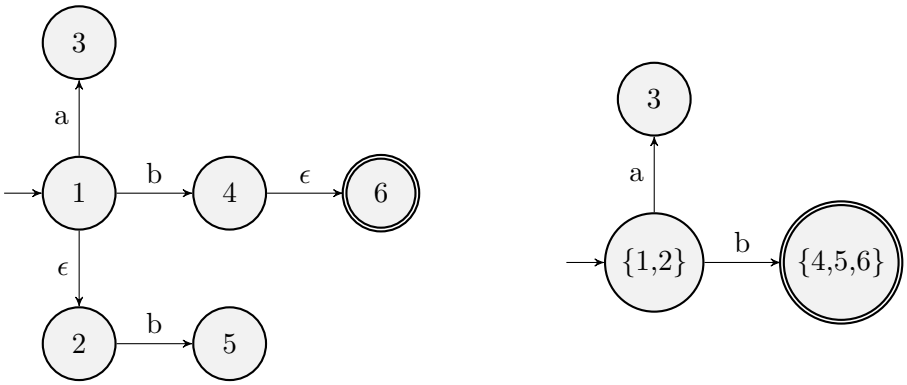



Figure 8.7: Converting an NFA to a DFA, accounting for $\epsilon$-transitions and multple edges with the same symbol

# Class 9

## 9.1 ML-Lex

Notes in brief:

Make the sources.cm file should explicitly include sml-nj and the basis library. Mine reads:

```
Group is

$/smlnj-lib.cm
$/basis.cm
driver.sml
errormsg.sml
tokens.sig
tokens.sml
tiger.lex
```

Otherwise, the only file you should need to modify is `tiger.lex`.

Second, states must have a semicolon at the end:

```
%%
%s COMMENT;
%%
```

Third, special variables like yypos and YYBEGIN are only available in the third section of an ML-Lex specification. This can result in some ugly/wordy lexer rules.

Fourth, use `ref` variables to implement global counters. These are mutable data types, most other data types in ML are not mutable (including variables defined). For example:

```
    val commentLevel = ref 0;
```

Access the value of a ref with the exclamation point and assign with `:=`

```
    commentLevel := (!commentLevel - 1);
```

See the textbook and the ML-Lex online documentation for examples in how to write rules. In general, a rule looks like:

```
if      => (Tokens.IF(yypos, yypos+2));
":="    => (Tokens.ASSIGN(yypos, yypos+2));
```

These two rules match literal expressions. The rule on the left hand side can also be a regular expression, such as:

`[a-zA-Z0-9]`$^{+}$ `=> (Tokens.ID(yytext, yypos, yypos+size(yytext)));`

See the file `tokens.sig` to see a list of all the defined tokens in the Tiger language.

## 9.2 Class 9 Exercises

# Class 10

Finite automata are insufficient to express syntax- we need a more powerful model!

## 10.1 Finite Automata are Insufficient for Syntax

The classic shortcoming of finite automata for recognizing syntax is the problem of recognizing *balanced* sets of parentheses. Consider constructing a machine that would try to recognize the language $\{a, (a), ((a)), (((a))), etc.\}$. If we try to come up with a DFA for such a language, we might start with something like Figure 10.1. This machine does indeed recognize every string in the language, but it also recognizes unbalanced sets of parentheses as well, such as $((a))))$ and $((((a)$. This machine has no "memory" or other way to recognize a specific set of parentheses.

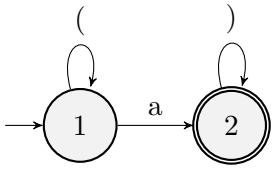



Figure 10.1: A DFA that recognizes unbalanced sets of parentheses.

Our only option for "counting" the number of parentheses is to use states. The machine in Figure 10.2 matches exactly zero or one set of balanced parentheses, or the language $\{a, (a)\}$. In order to match more parentheses we need to add additional states.

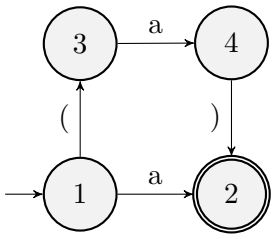



Figure 10.2: A DFA that recognizes up to one set of balanced parentheses.

Adding more states to the previous machine might yield something like Figure 10.3. However, the problem is soon obvious. This new machine can recognize the language

$\{a, (a), ((a)), (((a))), ((((a))))\}$, but those are the only strings it recognizes. To match longer sets of balanced parentheses, we'd need a machine with more states. To match arbitrarily many balanced parentheses, we would need arbitrarily many states, and unfortunately these are *finite automata* and not *infinite automata*.

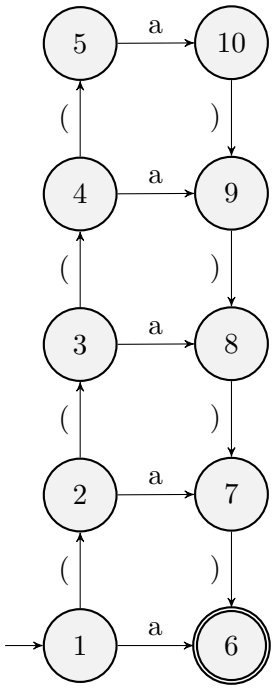



Figure 10.3: A DFA that recognizes up to four sets of balanced parentheses.

## 10.2 Context-Free Grammars

We need something more powerful to recognize sets of balanced parentheses and other syntactic constraints. This comes in the form of context-free grammars. Like finite automata, CFGs are said to *recognize*, *derive*, or *produce* a language of strings. Many of our previous definitions apply. A *language* is a set of strings. A *string* is a sequence of *symbols* drawn from an *alphabet*. The basic goal of parsing is to understand the syntactic structure of a program that is returned from the lexer, so generally our alphabet will be the set of lexer tokens.

A grammar is described as a series of *productions* between symbols. These are described with arrow notation as such:

$$S \to S; S$$

This production takes a symbol S and produces the string "S ; S". All grammars start

with a given *start symbol*, symbols are said to be *terminal* or *nonterminal* depending on whether they produce new symbols or not.

Consider the following grammar:

$$S \to S + S \qquad S \to S * S \qquad S \to Num$$

If we allow *Num* to represent natural numbers, then this grammar produces strings such as: "5 + 10" and "3 + 2 * 4".

This grammar is perfectly fine for describing strings, but in the context of compilers we want to use grammar productions to analyze syntactic structure to our programs. For this reason, it is desriable for a grammar to be *unambiguous*. An *ambiguous* grammar is one where strings may have multiple derivations, while unambiguous grammars only permit a single derivation. For example, we can derive the string "3 + 2 * 4" by applying the "S + S" rule and then applying the "S * S" rule, or we can apply them in the reverse order.

Each derivation has an associated *parse tree* that shows the sequence of productions that derive the string. For example, the string "3 + 2 * 4" has two possible parse trees:


Figure 10.4: Ambiguous derivation of the string "3 + 2 * 4"

Ambiguous grammars are problematic, because we would like to use CFGs to recognize and provide syntactic structure to our programs. The left-hand derivation could be interpreted as "3 + (2 * 4)" and the left-hand derivation could be interpreted as "(3 + 2) * 4". Unfortunately, these two statements have different values- 11 and 20- and so we don't get the syntactic structure for "free."

## 10.3 Unambiguous Grammars

Thankfully, it is often possible to construct an unambiguous grammar from an ambiguous grammar. In the example from the last section, the problem is that we have a choice of applying the addition or multiplication rule first, and depending on the order we apply

those rules, we get two different parse trees. However, we can transform that grammar so that, for example, we always have to apply addition before we apply multiplication. Taking as example from page 44 of Appel, the following grammar enforces this rule:

$$E \to E + T \qquad T \to T * F \qquad F \to Num \qquad E \to T \qquad T \to F$$

As a result, there is a single unambiguous derivation of the string, starting with addition and then doing multiplication. If we were to parse this tree from the bottom up, this would then also enforce the action of multiplication before subtraction, one of our normal arithmetic order of operations rules.

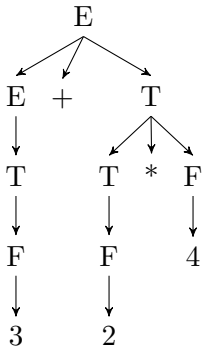



Figure 10.5: Unambiguous derivation of the string “3 + 2 * 4”

# Predictive Parsing, FIRST Sets

Parsing is the practice of imposing syntax on a sequence of tokens generated from our lexer. We use grammars to describe the valid syntax of programs, so in general the job of a parser is to decide which grammar productions generate an input program. This lets a complier understand the meaning of a program in a rigorous and mechanical way.

We will see three parsing techniques in Appel's book: *recursive descent parsing*, *LL* parsing, and *LR* parsing. Why three techniques? The difficulty of interpreting a grammar is related to the structure of a grammar- recursive descent is powerful in the sense that it can recognize more grammars than LL or LR parsing can, but where applicable LL or LR parsing could yield a more efficient. Since this is a property of the grammar, and not the algorithm, it's worth knowing different parsing techniques, as different languages might be more suited to one technique over another. Note also that there are other parsing algorithms beyond these three, but we will not cover them here.

## 11.1 Predictive Parsing

Let's formalize the parsing problem. The input to the parser is a sequence of tokens, and each token is a terminal symbol of a given grammar. Given a token, the parser must decide which grammar rule produced that particular token.

As the name implies, recursive descent parsing relies on the recursive structure inherent in grammar productions. When a grammar rule is identified, we can continue processing it by recursively calling code to handle parsing the subexpressions of the parent expression. For example, consider an if-statement:

> `if` *logical-expression* `then` *statement* `else` *statement*

Or, as might be expressed as a grammar production:

> $S \rightarrow$ if $E$ then $S$ else $S$

Assuming that no other grammar productions produce the "if" token, then as soon as we see an "if" we know immediately what the following tokens should be. In other words, we can predict what tokens come next and we know which grammar rules produced them. If our parser had two functions, `parse_E()` and `parse_S()`, then as soon as we see the

token “if” we know we need to call these other functions recursively. For example, our parser might include the following code:

```
fun parse_S = case token
 of IF => (eat(IF); parse_E(); eat(THEN); parse_S(); eat(ELSE); parse_S())
```

Note the purpose of the `eat()` function- the input stream is a global object shared by all recursive functions. Calling `eat(IF)` consumes the IF token from the input stream, and advances the token variable to point at the next input token. Therefore, when `parse_E()` is called, it will evaluate the next token after IF.

The above strategy relies on our ability to immediately tell which grammar production applies to a given statement. This property is not true of all grammars, for example consider the following:

$$E \to E + T \qquad E \to E - T \qquad E \to T \qquad T \to Num$$

It is not possible to write code for parsing this grammar like we have done above:

```
fun parse_S = case token
 of ?? => (parse_E(); eat(+); parse_T())
  | ?? => (parse_E(); eat(-); parse_T())
  | ?? => parse_T()
```

Nothing about the grammar distinguishes the three $E$ productions like the IF token did for the previous grammar. It is impossible to know whether a given $E$ production is referring to addition or subtraction without looking further ahead in the input stream. It is actually this property that distinguishes predictive parsers from general recursive descent parsers.

A *predictive parser* is always able to determine what grammar rule is being used with *one-symbol lookahead*. Predictive parsers admit a simple parsing strategy of the kind outlined above- consume one token at a time, with an if-then statement switching how to process each new token. Not all grammars admit this strategy, but when one does, a hand-written predictive parser is simple, compact, and efficient.

## 11.2 First Sets

We will see more complex forms of parsing shortly- and there are even parser generators we will see as well. In the same way that Lex can generate a lexer from a set of regular

expressions, parser generators exist that can generate an executable parser. However, there are tradeoffs in using a parser generator, so it would be nice to know when a simple predictive parser can be used. We will formalize the notion of *one-symbol lookahead* parsing with FIRST and FOLLOW sets. The discussion of these sets will also lead into the motivation for our more robust LL and LR parsing methods.

GIven a string of terminal and nonterminal symbols $\gamma$, the *first set* FIRST($\gamma$) is the set of all terminal symbols that begin a string derived from $\gamma$. For example, in the grammar:

$$E \to E + T \qquad E \to E - T \qquad E \to (E) \qquad E \to T \qquad T \to Num$$

Then FIRST($E$) = {( , Num}. The production of $(E)$ is optional, so a terminal string may start with ( if that is the first production followed, otherwise the first terminal symbol encountered will be Num. Similarly, FIRST($E + T$) = {( , Num}, and FIRST($T$) = {Num}.

If two different productions $E \to ?$ in a grammar have overlapping FIRST sets, then the grammar cannot be parsed with predictive parsing. This indicates a situation where the parser will not know immediately which grammar production is used with one-symbol lookahead. For example, FIRST($E + T$) = {( , Num} and FIRST($E - T$) = {( , Num}. Both $E + T$ and $E - T$ are productions of $E$, so there must be some valid program whose production cannot be immediately distinguished without lookahead. However, FIRST( $(E)$ ) = { ( } and FIRST($T$) = {Num}, so we can distinguish these productions with one-symbol lookahead.

To make this concrete, consider the program $5 + 3$. When the parser starts, it will be looking at the first token, which is 5/Num. We can't know whether the add or subtract production is used without looking ahead to the next token, where the difference between symbols "+" and "-" will unambiguously identify the production. In contrast, consider the program (7). The parser starts by looking at the ( token, which automatically identifies the first production rule used ($E \to (E)$) and rules out the $E \to T$ production.

If every production is unambiguously identified by nonterminal symbols then the grammar can be parsed with a predictive parser. Otherwise, there exists some terminal symbol $I$ in FIRST($\gamma_1$) and FIRST($\gamma_2$) that results from the productions $X \to \gamma_1$ and $X \to \gamma_2$, and a one-symbol lookahead parser will not know what to do with $X$.

See Appel chapter 3.2 for the computation of FIRST sets.

# FOLLOW Sets, Predictive Parse Tables

We look at how to implement parse tables efficiently.

## 12.1 Follow Sets

FOLLOW($X$) is the set of terminal symbols that can immediately follow X. For some nonterminal $t$, we say $t$ is in the FOLLOW set if any sequence of productions can produce $Xt$. This can be tricker to compute than it might seem at first, because some nonterminals are *nullable*. For example, in Appel's Grammar 3.12, $a$, $c$, and $d$ are in FOLLOW($X$) because:

$$Z \to XYZ \to XYXYZ$$

But then, we have several options, because X and Y are nullable:

$$XYXYZ \to Xd$$

$$XYXYZ \to Xad$$

$$XYXYZ \to Xcd$$

## 12.2 LL Parsing and Parsing Tables

*LL parsing* stands for *left-to-right parsing, leftmost derivation.* Recall that the input to the parser is a sequence of terminal symbnols from the lexer. In LL parsing, the input is consumed from left to right, and at each step the parser decides what grammar rule produces (derives) the leftmost set of terminal symbols. LL(1) parsing is LL parsing with one-symbol lookahead, meaning that only single leftmost symbol is considered. LL(2) is two-symbol lookahead and uses the two leftmost symbols, and this definition can be extended to LL(3), LL(4), and LL(K) lookahead. Doing so requires expanding our concept of our FIRST set to describe the first $k$ symbols of the input stream, but this is not difficult.

Note that LL(1) is equivalent to our predictive parser. Note also that every LL(1)

grammar is also an LL(2) grammar or an LL(3) grammar, and an LL(2) grammar is also an LL(3) grammar, etc.

Implementing LL parsing efficiently in practice is done with a lookup table based on FIRST and FOLLOW sets. This table has nonterminal symbolsdown the side, and terminal symbols across the top. See Figure 3.14 in Appel for an example. Each grammar production generates one or more entries in the parsing table.

For each production $X \to \gamma$:

- Insert that production in row $X$ and column $T$ for each terminal $T$ in FIRST($\gamma$). For example, if our grammar includes production $A \to B$ and FIRST($B$) is the set $\{a, b, c\}$, then insert the production $A \to B$ at locations ($A$, $a$), ($A$, $b$), and ($A$, $c$) in the parse table.
- If $\gamma$ is nullable, insert that production in row $X$ and column $T$ for each terminal $T$ in FOLLOW($X$). For example, if our grammar includes the production $X \to Y$, where $Y$ is nullable, and FOLLOW($X$) is the set $\{a, b, d\}$, then insert the production $X \to Y$ at locations ($X$, $a$), ($X$, $b$), and ($X$, $d$) in the parse table. (See Appel Figure 3.14 for this example.)

## 12.3 Ambiguous vs. Unambiguous Parse Tables and Left Recursion

An unambiguous parse table has at most one entry in any given cell. This means for any given leftmost nonterminal sequence, there is only one production that the parser could take, and so the parser does not have to "guess" which derivation to use. If the parse table needs only one nonterminal, then an unambiguous parse table is LL(1) (or we could build a predictive parser with recursive descent). If the parse table needs two nonterminal symbols to be unambiguous, then the parse table is LL(2), etc.

In general, a practical parser must be unambiguous. Thus, given an ambiguous parse table, we might think about how to build an unambiguous parse table.

## 12.4 Left Factoring and Ambiguity

$S \to IfXthenYelseZ$

$S \to IfXthenY$

Is a problem but can become:

$S \to IfXthenYT$

$T \to$

$T \to elseZ$

| | a | c | d |
|---|---|---|---|
| X | $X \to a$<br>$X \to Y$ | $X \to Y$ | $X \to Y$ |
| Y | $Y \to$ | $Y \to$<br>$Y \to c$ | $Y \to$ |
| Z | $Z \to XYZ$ | $Z \to XYZ$ | $Z \to d$<br>$Z \to XYZ$ |

**FIGURE 3.14.** Predictive parsing table for Grammar 3.12.

Figure 12.1

# LR Parsing

Left-to-right parsing, rightmost derivation

## 13.1 LR Parsing Algorithm

Like with LL parsing, the grammar is converted into a parse table. However, using the parse table is somewhat more complicated. In LL parsing our parse table was indexed only by terminal symbols (across the top) and nonterminal symbols (down the side).

Our LR parse tables need to be slightly more complex. The parse table can be thought of as encoding a DFA, with states down the side and both terminals and nonterminals across the top. Each cell of the parse table specifies a state transition, and optionally can consume input or apply grammar reductions. We also maintain a *stack* while LR parsing, and this stack tracks the current state of the DFA as well as current input symbols.

Executing an LR parse requires four operations given by the parse table in Appel Table 3.19 / Figure 13.2:

- Shift N - Push the next input symbol onto the stack and push state number $n$ onto the stack
- Goto N - Push the state number $n$ onto the stack
- Reduce K - The reduce operation is complex and has multiple steps:
    1. Eat the top stack symbols (and associated state numbers) corresponding to grammar rule $k$
    2. Look at and remember the state number $p$ now on top of the stack
    3. Push the left hand side $X$ of grammar rule $k$ onto the stack
    4. Look up the column $X$ for state $p$ and apply the Goto operation found there
- Accept - Parsing terminates and the program is a valid program according to the parse table

For example, let's parse the program "a := 7; b := c + d" with Appel's grammar 3.1 and Appel's parse table 3.19 / Figure 13.2. We can observe the action of the parser by watching the stack and observing which rules are applied:

| | *Stack* | *Input* | *Action* |
|---|---|---|---|
| 1 | $_1$ | a := 7; b := c + d $ | Shift a |
| 2 | $_1$ $id_4$ | ~~a~~ := 7; b := c + d $ | Shift := |
| 3 | $_1$ $id_4$ $:=_6$ | ~~a :=~~ 7; b := c + d $ | Shift 7 |
| 4 | $_1$ $id_4$ $:=_6$ $num_{10}$ | ~~a :=~~ ~~7~~; b := c + d $ | Reduce num→ E, Rule 5 |
| 5 | $_1$ $id_4$ $:=_6$ $E_{11}$ | ~~a :=~~ ~~7~~; b := c + d $ | Reduce id := E → S, Rule 2 |
| 6 | $_1$ $S_2$ | ~~a :=~~ ~~7~~; b := c + d $ | Shift ; |
| 7 | $_1$ $S_2$ $;_3$ | ~~a :=~~ ~~7~~; b := c + d $ | Shift b |
| 8 | $_1$ $S_2$ $;_3$ $id_4$ | ~~a :=~~ ~~7~~; ~~b~~ := c + d $ | Shift := |
| 9 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ | ~~a :=~~ ~~7~~; ~~b :=~~ c + d $ | Shift c |
| 10 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $id_{20}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ + d $ | Reduce id→ E, Rule 4 |
| 11 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $E_{11}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ + d $ | Shift + |
| 12 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $E_{11}$ $+_{16}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ d $ | Shift d |
| 13 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $E_{11}$ $+_{16}$ $id_{20}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ ~~d~~ $ | Reduce id→E, Rule 4 |
| 14 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $E_{11}$ $+_{16}$ $E_{17}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ ~~d~~ $ | Reduce E + E → E, Rule 6 |
| 15 | $_1$ $S_2$ $;_3$ $id_4$ $:=_6$ $E_{11}$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ ~~d~~ $ | Reduce id := E → S, Rule 2 |
| 16 | $_1$ $S_2$ $;_3$ $S_5$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ ~~d~~ $ | Reduce S; S → S, Rule 1 |
| 17 | $_1$ $S_2$ | ~~a :=~~ ~~7~~; ~~b :=~~ ~~c~~ ~~+~~ ~~d~~ $ | ACCEPT |

Note the action of Reduce depends on the state prior to the consumed input. At step 4 the "$num_{10}$" token is consumed, and we look up nonterminal $E$ in state 6 to see the command Goto 11, so we push $E_{11}$ to the stack. Similarly, at step 5 the tokens "$id_4$ $:=_6$ $E_{11}$" are consumed, and we look up nonterminal $S$ in state 1 to see the command Goto 2, so we push $S_2$ to the stack.

1 $S \to S$ ; $S$
2 $S \to$ id := $E$
3 $S \to$ print ( $L$ )
4 $E \to$ id
5 $E \to$ num
6 $E \to E + E$
7 $E \to (S , E)$
8 $L \to E$
9 $L \to L , E$

**GRAMMAR 3.1.** A syntax for straight-line programs.

Figure 13.1

| | id | num | print | ; | , | + | := | ( | ) | $ | S | E | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | s4 | | s7 | | | | | | | | g2 | | |
| 2 | | | | s3 | | | | | | a | | | |
| 3 | s4 | | s7 | | | | | | | | g5 | | |
| 4 | | | | | | | s6 | | | | | | |
| 5 | | | | r1 | r1 | | | | | r1 | | | |
| 6 | s20 | s10 | | | | | | s8 | | | | g11 | |
| 7 | | | | | | | | s9 | | | | | |
| 8 | s4 | | s7 | | | | | | | | g12 | | |
| 9 | | | | | | | | | | | | g15 | g14 |
| 10 | | | | r5 | r5 | r5 | | | r5 | r5 | | | |
| 11 | | | | r2 | r2 | s16 | | | | r2 | | | |
| 12 | | | | s3 | s18 | | | | | | | | |
| 13 | | | | r3 | r3 | | | | | r3 | | | |
| 14 | | | | | s19 | | | | s13 | | | | |
| 15 | | | | | r8 | | | | r8 | | | | |
| 16 | s20 | s10 | | | | | | s8 | | | | g17 | |
| 17 | | | | r6 | r6 | s16 | | | r6 | r6 | | | |
| 18 | s20 | s10 | | | | | | s8 | | | | g21 | |
| 19 | s20 | s10 | | | | | | s8 | | | | g23 | |
| 20 | | | | r4 | r4 | r4 | | | r4 | r4 | | | |
| 21 | | | | | | | | | s22 | | | | |
| 22 | | | | r7 | r7 | r7 | | | r7 | r7 | | | |
| 23 | | | | | r9 | s16 | | | r9 | | | | |

**TABLE 3.19.** LR parsing table for Grammar 3.1.

Figure 13.2

## 13.2 Heirarchy of Parser Power

The difference between LL and LR parsing is leftmost derivation versus rightmost derivation. While this may seem like a small difference, it has far-reaching effects. The LL parse is somewhat simpler because the next derivation is uniquely determined by the current input symbol and the current nonterminal symbol. The LR parse, however, cannot immediately determine what grammar rule to apply without consuming several input symbols. Conceptually, this allows the LL parser to start with the grammar start symbol and then try to derive the input string, while LR parsers start with the input string, and then attempt to repeatedly reduce it back to the start symbol.

LR parsers are described as making shift-reduce decisions. LL parsers make first-follow decisions.

LL parsers are strictly less powerful than LR parsers. LR parsers are more powerful, but LR(1) parsers can have large parse tables- large enough that at the time they were concieved in the mid-60's, modern computers could not run them. As a result, several

attempts to get LR(1)-like behavior were made that did not have the same memory footprint. Notably, SLR, or Simple LR, is stronger than LR(0) but less strong than LALR(1), and LALR(1) is less powerful than LR(1). However, the memory footprint of SLR and LALR(1) are the same as LR(0), making them feasible for the time. Most practical languages are LALR(1) with the exception of ambiguity.

LR parsers give a bottom-up parse of a language, LL parsers give a top-down parse of a language.

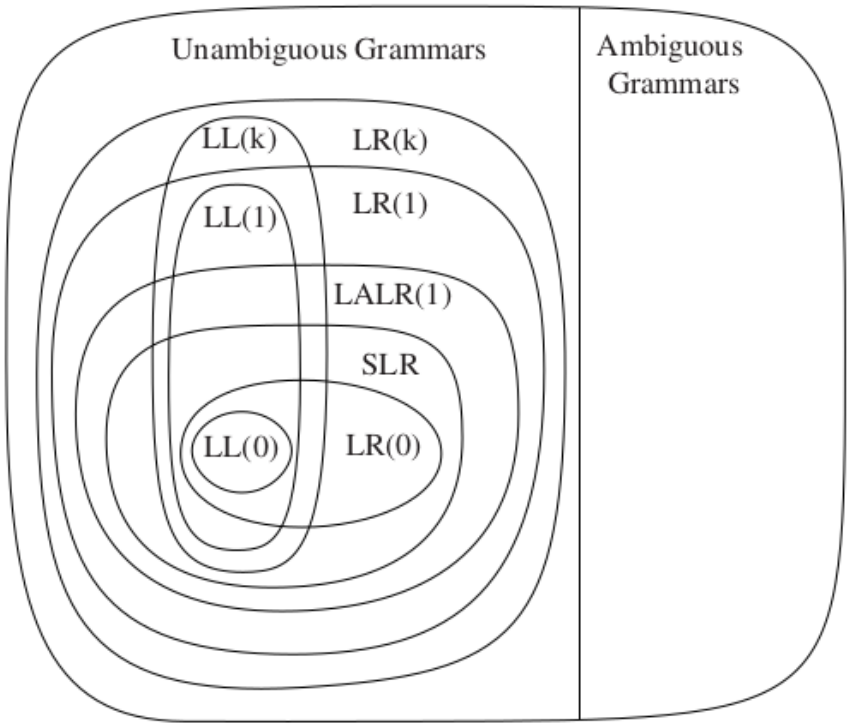



**FIGURE 3.29.** A hierarchy of grammar classes.

Figure 13.3

## 13.3 Ambiguity in LR parsing

E * E

E / E

E + E

E - E

# Program 3 - Parser

Defining a grammar for the Tiger programming language, and building a parser with ML-Yacc. Appel gives a formal description of the problem assignment on pages 81-82, but most of section 3.4 *Using Parser Generators* is relevant and I would suggest you read it carefully before starting.

## 14.1 ML-YACC

Like with ML-Lex, ML-Yacc is an ML version of Yacc: Yet Another Compiler Compiler. Both programs use /emphdeclarative specifications and generate executable programs. ML-Lex takes a set of regular expressions to generate an executable lexer. ML-Yacc takes a grammar and generates an LR parse table and executable parser.

The main ML-Yacc specification file is `tiger.grm`, which defines the grammar the parser is supposed to recognize. The file format is similar to ML-Lex's `.lex` file: three sections that are separated by %% tokens:

```
user declarations
%%
parser declarations
%%
grammar rules
```

Like with ML-Lex, viewing the ML-Yacc manual online may be helpful.

## 14.2 Running the Parser

The `sources.cm` file needs to be modified so SML knows to look in the installation directory for system libraries. My `sources.cm` file is:

```
Group is
```

```
tiger.lex
errormsg.sml
parsetest.sml
tiger.grm
$/smlnj-lib.cm
$/basis.cm
$/ml-yacc-lib.cm
```

Otherwise, the only file you need to modify is `tiger.grm`, and mainly only the third section (the grammar rules). The file `parsetest.sml` gives a driver for running the parser, and the file `errormsg.sml` provides error reporting support.

Addtionally, Appel provides his `tiger.lex.sml` file, which is his lexer built in Program 2. If you would like to use your own lexer from Program 2 you can replace this file with the one built from your own `tiger.lex` specification.

## 14.3 Output

When the `sources.cm` file is loaded it will automatically run ML-Yacc, build an LR parse table, and build the parser. At this point you may get an error if your grammar specification is malformed. You may also see warning messages about reduce/reduce or shift/reduce conflicts, such as:

```
5 reduce/reduce conflicts
8 shift/reduce conflicts
```

See Appel sections 3.3 and 3.4 for more on shift/reduce and reduce/reduce conflicts. In general, your grammar should not have any reduce/reduce conflicts and only have shift/reduce conflicts for well-known cases such as the ambiguity of add/multiply or the ambiguity of multiple if/else statements. Resolving conflicts is described in sections 3.3 and 3.4.

After loading `sources.cm` the parse table has been constructed. You can review this parse table in the file `tiger.grm.desc`. The parse table explicitly flags states with unhandled conflicts as errors. If you are not sure about how your grammar operates, or why a conflict is occuring, it may be helpful to hand-simulate your parse table.

Lastly, you can invoke the parser in the SML interpreter by typing `Parse.parse "test.tig";`. If successful, your parser will produce no output, indicating that it was able to reduce the

entire program back to the start symbol. If unsuccessful, you will get a `syntax error` message along with the parser describing what it thinks the error is. For example, if you give the input file "`var x := 10`" but don't have a grammar rule that reduces "`VAR ID ASSIGN`" the parser reports "`test.tig:2.1:syntax error: deleting VAR ID ASSIGN`"

## 14.4 Ambiguity

We have seen several examples in class and the text where it is difficult to avoid ambiguous grammars. These are examples such as:

```
10 + 20 * 30
```

and

```
if E then if E then S else S
```

These cases are ambiguous, and writing a fully unambiguous grammar might be difficult or impossible. See section 3.3 *LR Parsing of Ambiguous Grammars* for some ideas of what to do here. See also section 3.4 *Conflicts* and *Precedence Directives* for pointers on how to handle these cases in ML-Yacc.

## 14.5 Examples

There are a number of grammar examples in sections 3.3 and 3.4 for you to use as inspiration when writing your own grammar for Tiger. See specifically Grammar 3.30, Grammar 3.31, Grammar 3.35, and Grammar 3.36, but others may be helpful as well.

## 14.6 Partial Output

The SML clauses in your final grammar should all return the unit `()`, as seen in examples Grammar 3.31 and Grammar 3.36. However, you can place SML `print()` statements in here if you feel the need to observe how your parser is working. For example (though this is not necessarily a good starting point for your grammar):

```
stm     : stm stm                       (print("S -> S S\n"))
        | VAR ID ASSIGN exp             (print("S -> var ID ASSIGN exp\n"))
```

These statements will print when the parser reduces according to these rules. Remember that LR parsing provides a bottom-up parse, so the outermost leaf expressions will be reduced first, followed by higher level expressions. For example, applying the above grammar to the file:

```
var x := 10
var y := 20
var z := 30
```

will result in the output:

```
- Parse.parse "test.tig";
[autoloading]
[autoloading done]
S -> var ID ASSIGN exp
S -> var ID ASSIGN exp
S -> var ID ASSIGN exp
S -> S S
S -> S S
val it = () : unit
```

because each assignment statement is reduced to $S$ first via rule 2 and then the string $SSS$ is reduced to $S$ via rule 1.

# Abstract Syntax

## 15.1 Semantics

A lexer ensures lexical correctness and a parser ensures syntactic correctness.

Lexical correctness:

- The file only contains proper input tokens
- Identifiers follow identifier rules
- Strings, comments, etc. are properly terminated

Syntactic Correctness:

- Grammar rules are satisfied
- Expressions have correct structure

What is left for the semantic analyzer to do? A lot, actually!

Examples of semantic problems:

- Cannot use values before they're defined
- Functions have to be declared/defined before being used
- Can't redefine functions/variables after they've been defined
- Operation types have to be compatible (multiply float by int, or string by int)

Just because a program is well-formed does not mean it has *semantic* meaning.

For example- people suffering from Wernicke's aphasia sometimes use good syntax, but cannot produce semantically meaningful sentences. They may not even realize they're

speaking differently, and can get frustrated easily. They sound like they're speaking fluently, but don't make sense. See an example: https://www.youtube.com/watch?v=3oef68YabD0

Another classic example is the sentence "Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo." This is a grammatically correct sentence in English that is terribly ambiguous. For the purpose of programming languages, we want to constraint the set of all grammatically correct programs to the set of unambiguous, semantically meaningful programs. https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo

## 15.2 An un-semantic program

Does the following program look reasonable?

```
void myFunc( int x, int y ){
    int z = "Hello, world!"
    return (x + y) * z;
}

int main(){
    int a = myFunc() + b;
}
```

The program is lexically and syntactically correct, but has no semantic meaning.

- Is z supposed to be an integer or a string?
- If z is a string, we can't multiply integers by strings.
- The function myFunc() attempts to return a value, but has return type void.
- The return value of myFunc() is used in main(), but myFunc() does not return a value.
- The variable b is undefined.

## 15.3 Where should semantic analysis be done?

It is possible to do semantic analysis inside our parser- such compilers can compile a program in a single pass. For example, an ML-Yacc grammar rule might contain:

```
exp :   INT             ( INT )
    |   exp PLUS exp    ( exp1 + exp2 )
    |   exp TIMES exp   ( exp1 * exp2 )
```

If the first expression is given the name exp1 and the second expression is given the name exp2, then the ML language closures to the right of each grammar rule can have the effect of executing the semantic intent of each grammar rule. It is possible to write a semantic analyzer that only fits within the ML phrases of an ML-Yacc parser.

However, such software is

- Difficult to maintain
- Constrains the semantic analyzer to process the program in the order that it is parsed

## 15.4 Parse Trees

It is better to separate the process of parsing from the process of semantic analysis. To do so, we need to pass a semantic representation of the parsed program from the parser to the semantic analyzer. This is the purpose of the Abstract Parse Tree.

A *parse tree* is a tree data structure that has one leaf node for each terminal token of input, and one internal node for each grammar rule reduced during the parse.

We have seen already that the action of the parser can build an implicit parse tree. An LL parser starts with the starting nonterminal symbol and attempts to derive the whole program top-down, applying grammar productions as it goes. An LR parser starts with the output program and attempts to reduce back to a single starting nonterminal symbol- identifying which grammar productions are needed as it goes.

Such a parse tree- derived from the action of a grammar in a parser- is called a *concrete parse tree.*

Concrete parse trees can be useful, but recall all of the little issues we had to solve to get a functional LR(1) or LALR parser working. Such parsers have to deal with ambiguity, and as a result introduce grammar rules that don't nicely correspond to semantic actions. For example, in dealing with left-recursion or doing left factoring we introduced extra nonterminal symbols. Recall, the rules:

$E \rightarrow E + T$

$E \rightarrow T$

These lead to unwanted left recursion, and such a language cannot be LL(1). To eliminate the left recursion we re-wrote the grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow$

In this second form, the nonterminal $E'$ has a tricky semantic action- just by looking at $E$ or $E'$ it is not clear what the semantic action should be because the plus operator belongs to the next nonterminal in the sequence.

## 15.5 Abstract Syntax Trees

Rather than working with the concrete syntax directly, it would be preferable to deal with *abstract syntax*, and the data structure of choice for this in a compiler is an *abstract syntax tree*, or AST. An abstract syntax tree cleanly conveys the original phrase structure of the input program, without complications due to the necessities of parsing, and without doing any semantic interpretation.

The grammar of the AST does not need to match the concrete grammar that is used by the parser. For example, there is no need for the abstract syntax to be unambiguous. The parser grammar can concretely parse a program and provides a specific execution order for statements in the program. The AST can inherit that unambiguous execution order.

In practice, the AST is built during parsing alongside the concrete parse of the program. Parsers like ML-Yacc allow for each grammar reduction to execute some code. For Program 3 we left this code empty, but in Program 4 we will use these to build an AST while we are parsing.

See Program 4.5 in the book to see the direct implementation of semantics in a parser, such as might be suitable for use in an online interpreter such as Python. See Program 4.7 to see the construction of an AST from the action of a parser, such as might be suitable for use in a compiler.

# Semantic Analysis

Appel Chapter 5: *Semantic Analysis* connects variable definitions to their uses, checks that each expression has a correct type, and translates abstract syntax into a simpler representation suitable for generating machine code.

## 16.1 Identifier Binding

Whenever an indentifier is used in a program it must represent some concrete value. Variables must be mapped to a concrete value, function names must be mapped to code, types must be mapped to a concrete definition, polymorphic interfaces must be mapped to a concrete interface, etc. The tying of an identifer to a concrete instantiation is called *binding*.

Sometimes it can be tricky to resolve an identifier to an appropriate binding. Some languages support concepts such as object methods and namespaces, which add complication, but almost every language supports scoping rules. For example, the following program in C has four valid definitions of the variable `foo`:

```
#include <stdio.h>

int foo = 0;

int main(){

        printf("First value of foo:  %d\n", foo );

        int foo = 1;

        printf("Second value of foo: %d\n", foo );

        if ( 1 ){
                int foo = 2;
                printf("Third value of foo:  %d\n", foo );
```

```
                { //An unnamed closure
                        int foo = 3;
                        printf("Fourth value of foo: %d\n", foo );
                }
        }
}
```

When run, the program produces the output:

```
First value of foo:  0
Second value of foo: 1
Third value of foo:  2
Fourth value of foo: 3
```

Tracking these bindings is the job of *symbol tables*, which map identifiers to values. Each symbol table also describes an *environment*, and our author uses this term interchangably.

In the case of scoping rules, each scope can be described by a separate symbol table. For example, in the following C program there are two scopes, the global scope and the main scope:

```
int x = 5;
int y = 10;

int main(){
    int x = 20;
    int z = 50;
    ...
}
```

These two symbol tables can be represented with the symbol sigma:

$\sigma_{global} = \{x \to 5, y \to 10\}$

$\sigma_{main} = \{x \to 20, z \to 50\}$

Typically, the scoping rules in real programming languages say that the most recent scope is the highest priority. In the last program, the definition $x \to 20$ has priority inside the `main()` function, because it is the highest, most recent, narrowest scope. We can describe the combination of scopes as X + Y, where scopes to the right have priority

over scopes to the left. When a lookup occurs, the rightmost symbol table searches for the identifier, and if that fails the search spills over to the next symbol table in line. If the symbol is not found in any symbol table, this generates an undefined variable error.

Environments do not need to neatly nest the way we have seen above. In object-oriented languages there may be many environments being used simultaneously, and resolving an identifier correctly requires resolving the identifier through the correct set of symbol tables. Two different objects may have private variables with the same name, for example. Or, one object might define a private variable with the same name as a global variable, but another might not. If our objects are called A and B, then resolving an identifier in object A could require searching $\sigma_{global} + \sigma_A$ while resolving an identifier in object B requires searching $\sigma_{global} + \sigma_B$.

See the book for discussion of implementation of symbol tables, and for discussion of specific bindings in Tiger.

# Template

Class Intro

## 17.1 A section

Section text

### 17.1.1 A subsection

Subsection text