CO221 - Compilers

6th January 2020

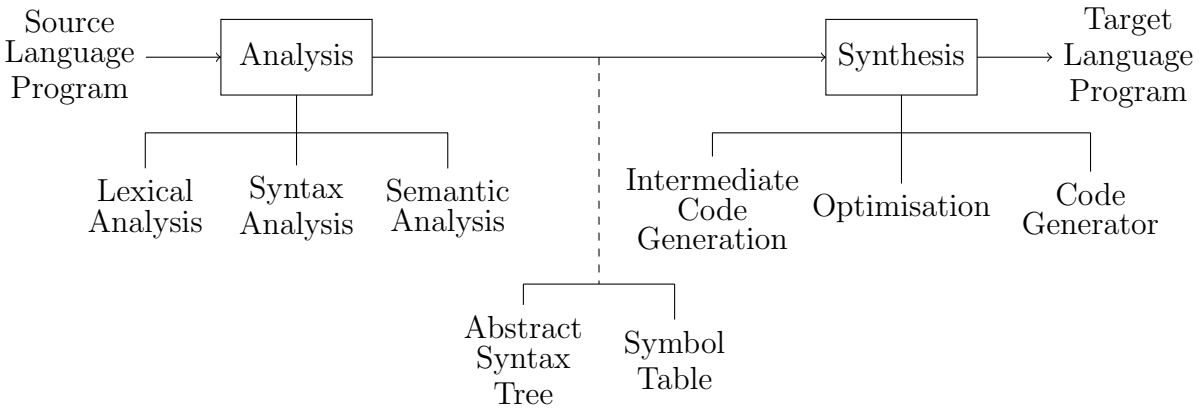
A compiler is a program which processes programs, including translating a program written in one language (usually higher level) to another programming language (usually in a lower level). In our course, the focus is to generate assembly code from the high level language. This translation goes between high level human concepts, and the data manipulation the machine performs.

Structure

The general structure of a compiler is as follows;

1. **input** takes in an input program in some language
2. **analysis** constructs an internal representation of the source structure
3. **synthesis** walks the representation to generate the output code
4. **output** creates an output in the target language

In more detail, it can be represented as follows;



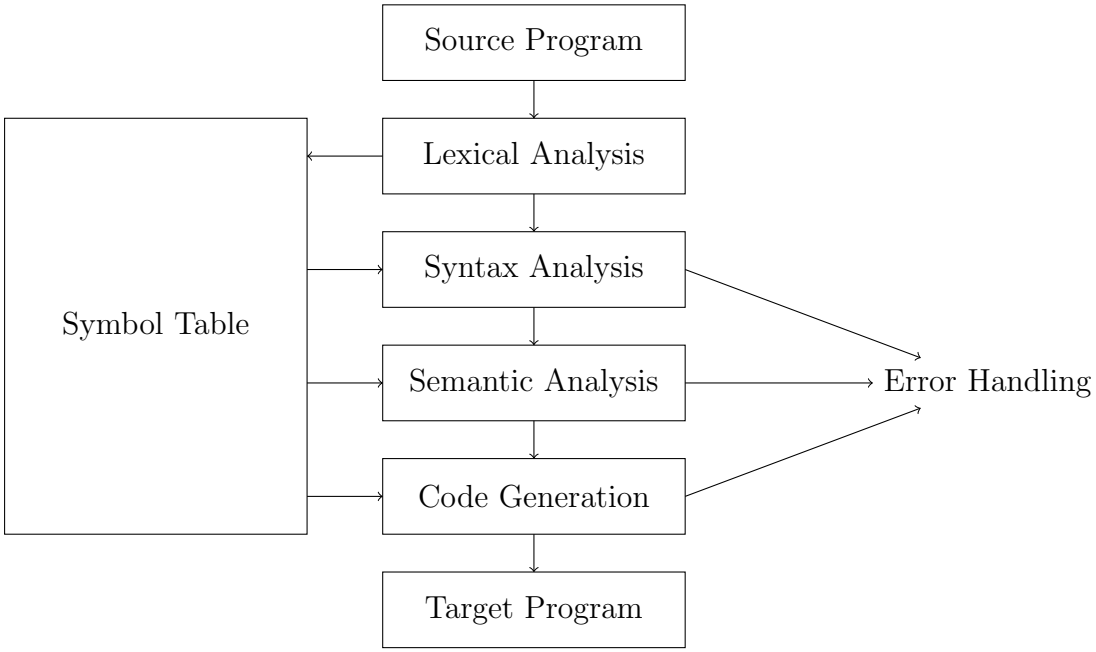
- **lexical analysis** looks at characters of input program, analyses which are keywords (such as converting `if`, and `while` to corresponding tokens), which are user defined words, and which are punctuation, etc.
- **syntax analysis** discovers structure of input
- **semantic analysis** checks that variables are declared before they are used, and that they are used consistently with their types etc.

Simple compilers go straight to code generation, but optimising compilers do several passes of intermediate code generation and optimisation.

The symbol table holds data on variables, such as types. Sometimes we need to know the type of the variable, in order to generate code for the variable, for example if we were to print a variable, it would need to generate different code for strings than it would need to do for integers. Scope rules are also needed.

Phases

Whether all of these phases are done in the order shown is a design choice. For example, lexical analysis and syntax analysis are often interleaved. This can be done when the syntax analysis stage needs the next symbol, and therefore the lexical analysis stage can be used.



Syntax Analysis

This is also known as parsing. Languages have a grammatical structure specified by grammatical rules in a **context-free grammar** such as BNF (**Backus-Naur Form**). The output of the analyser is a data structure which represents the program structure; an **abstract syntax tree**. The writer of the compiler must design the AST carefully such that it is easy to build, as well as easy to use by the code generator.

A language specification consists of the following;

- **syntax** grammatical structure

  in order to determine that a program is syntactically correct, one must determine how the rules were used to construct it
- **semantics** meaning

For example, we can encode the rules for a statement as follows (anything in quotes is a terminal), in BNF;

$$\text{stat} \rightarrow \text{'if' '(' exp ')' stat 'else' stat}$$

Each BNF production is a valid way for a non-terminal (LHS) to be expanded (RHS) into a combination of terminals and non-terminals. Only terminals can appear in the final results (they are lexical tokens).

To prove the following is a valid example of stat, we'd need to show that a can be derived from exp, and that both b and c can be derived from stat.

$$\text{if (a) b else c}$$

Context-Free Grammars

Formally, a context-free grammar consists of the following four components;

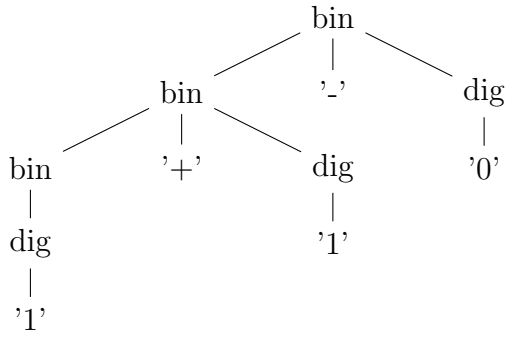
- S a non-terminal start symbol
- P a set of productions
- t a set of tokens (terminals)
- nt a set of non-terminals

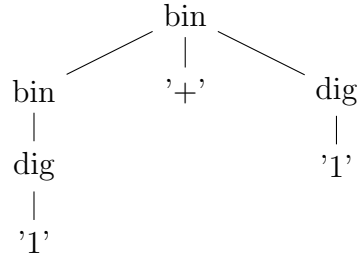
For example, consider the following BNF, and their associated components;

$$\begin{aligned}
&\text{bin} \rightarrow \text{bin '+' dig} \mid \text{bin '-' dig} \mid \text{dig} \\
&\text{dig} \rightarrow \text{'0'} \mid \text{'1'} \\
&t = \{\text{'+'}, \text{'-'}, \text{'0'}, \text{'1'}\} \\
&nt = \{\text{bin}, \text{dig}\} \\
&S = \text{bin}
\end{aligned}$$

A string of only terminals (**sentential form**) can be derived using the grammar by beginning with the start symbol, and repeatedly replacing each non-terminal with the RHS from a corresponding production. We refer to the set of all sentential forms derived from the start symbol as the **language** of a grammar.

We can prove that some string is in the language of a grammar by constructing a **parse tree**. For example, to prove that "1+1-0" $\in L(G)$, and "1+1" $\in L(G)$ we can use the following trees;

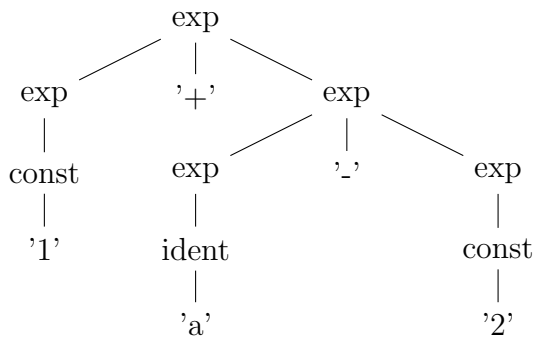


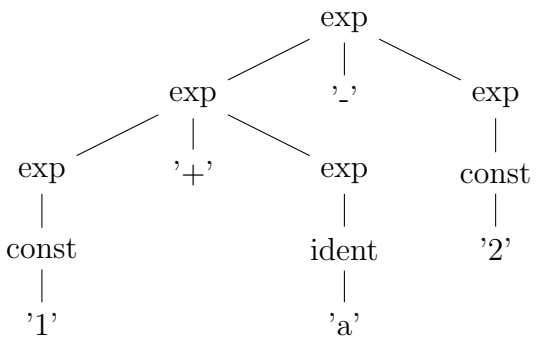


Ambiguity

A grammar is referred to as **ambiguous** if its language contains strings which can be generated in two different ways. Essentially, there exists some string in $L(G)$ which has two different parse trees. Consider string "1 + a - 3" in the following grammar, and the parse tree(s) associated;

$$\text{exp} \rightarrow \text{exp '+' exp} \mid \text{exp '-' exp} \mid \text{const} \mid \text{ident}$$





While the string is still valid, and in the language, our issue is with the ambiguity, as we want to generate a program uniquely. The reason our grammar is broken is due to the recursive use of the non-terminal exp on both sides, which means we're given a choice of which side to expand when generating.

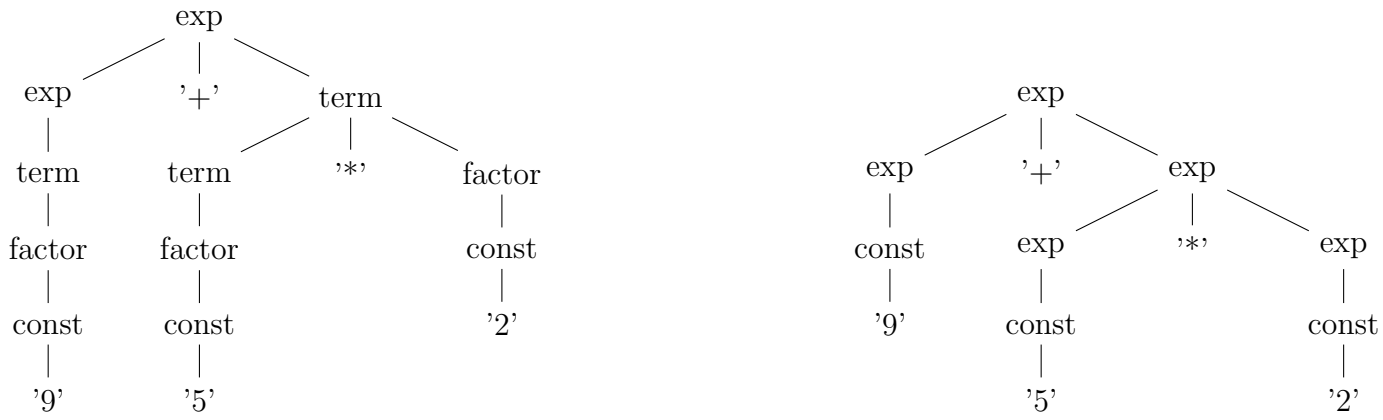
Associativity and Precedence

For our example language, we're using all left-associative operators. We also want to maintain that '*' and '/' have higher precedence than '+' and '-'. One way of doing this is to split the grammar

into layers, by having separate non-terminals for precedence levels. This method can be done with the following unambiguous grammar for arithmetic expressions;

$$\begin{aligned} \text{exp} &\rightarrow \text{exp } \text{'+'} \text{ term} \mid \text{exp } \text{'-'} \text{ term} \mid \text{term} \\ \text{term} &\rightarrow \text{term } \text{'*'} \text{ factor} \mid \text{term } \text{'/'} \text{ factor} \mid \text{factor} \\ \text{factor} &\rightarrow \text{const} \mid \text{ident} \end{aligned}$$

Now, we can unambiguously generate the parse tree (and thus the unique abstract syntax tree) for "9+5*2";



It's important to note that the **abstract** syntax tree doesn't need this in contrast, as only the parse tree needs it.

Parsers

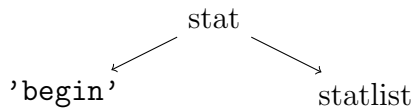
The parser checks that the input is grammatically correct, and builds an AST representing the structure. In general, there are two classes of parsing algorithms;

- **top-down / predictive** — we are using recursive descent
- **bottom-up** — also known as shift-reduce

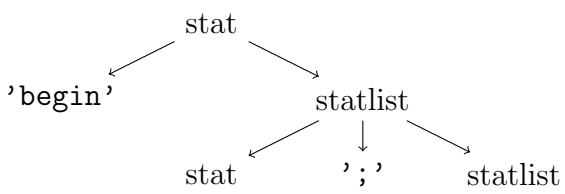
For this, we will use the input "`begin S; S; end`", with the following grammar;

$$\begin{aligned} \text{stat} &\rightarrow \texttt{'begin'} \text{ statlist} \mid \texttt{'S'} \\ \text{statlist} &\rightarrow \texttt{'end'} \mid \text{stat } \texttt{';'} \text{ statlist} \end{aligned}$$

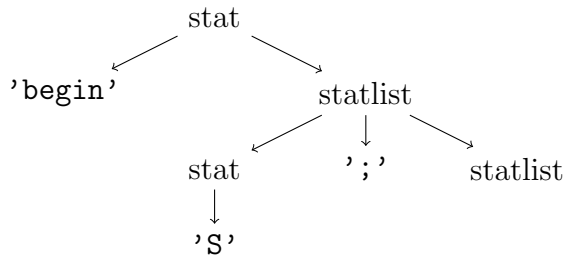
When we start top-down parsing, we start with the non-terminal stat. The first token we identify is the `'begin'`, thus our tree becomes the following;



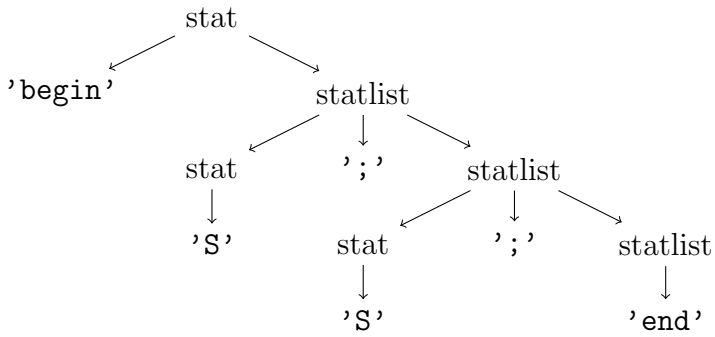
However, as the next symbol isn't the terminal `'end'`, we have to use an alternative. As we only have one alternative, we can predict it, and thus the tree becomes;



As the next symbols are the terminal `'S'`, and the terminal `';'`, we can tick them off, thus the tree becomes;



This process continues, until we reach the final tree;



On the other hand, bottom-up parsing tries to use all the RHSs (whereas top-down tries to match a non-terminal by trying each of the RHSs), and replaces it with a non-terminal, by using the production in reverse. Bottom-up succeeds when the whole input is replaced by the start symbol.

In general, we push the current symbol onto the stack (or the reduction if we can reduce it). For example, once we encounter `'S'`, we can reduce it to stat, and similarly once we encounter `'end'`, we can reduce it to statlist.

stack	current symbol	remaining tokens	S/R	note
	'begin'	'S' ';' 'S' ';' 'end'	S	nothing to do yet
'begin'	'S'	';' 'S' ';' 'end'	R	terminal for stat
'begin'	stat	';' 'S' ';' 'end'	S	no more work
'begin' stat	';'	'S' ';' 'end'	S	nothing to do yet
'begin' stat ';'	'S'	';' 'end'	R	terminal for stat
'begin' stat ';'	stat	';' 'end'	S	no more work
'begin' stat ';' stat	';'	'end'	S	nothing to do yet
'begin' stat ';' stat ';'	'end'		R	terminal for statlist
'begin' stat ';' stat ';'	statlist			
'begin' stat ';' stat ';'	statlist		R	match for statlist
'begin' stat ';'	statlist			
'begin' stat ';'	statlist		R	match for statlist
'begin'	statlist			
'begin'	statlist		R	match for stat
	stat			complete

Simple Compiler in Haskell

If the input to the parser is a simple string of characters, representing an arithmetic expression, following the BNF defined below;

expr → fact '+' expr | fact
fact → number | identifier

The string `a + b + 1` would become the following sequence of tokens, after lexical analysis;

```
[IDENT "a", PLUS, IDENT "b", PLUS, NUM 1]
```

It's important to note that this is a right-recursive grammar, as a recursive descent method **will not** work with left-recursive grammars.

```
1  data Token
2    = IDENT [Char] | NUM Int | PLUS
3  data Ast
4    = Ident [Char] | Num Int | Plus Ast Ast
5    deriving (Show)
6  data Instr
7    = PushVar [Char] | PushConst Int | Add
8    deriving (Show)
9  parse :: [Token] -> Ast
10 parse ts =
11   let (tree, ts') = parseExpr ts
12   in case ts' of
13     [] -> tree
14     _  -> error "Excess tokens"
15 parseExpr :: [Token] -> (Ast, [Token])
16 parseExpr ts
17   = let (factTree, ts') = parseFact ts
18     in case ts' of
19       (PLUS : ts') ->
20         let (sExpTree, ts'') = parseExpr ts'
21         in (Plus factTree sExpTree, ts'')
22       other -> (factTree, other)
23 parseFact :: [Token] -> (Ast, [Token])
24 parseFact (t:ts)
25   = case t of
26       NUM n -> (Num n, ts)
27       IDENT x -> (Ident x, ts)
28       _ -> error "Expected a number or identifier"
29 translate :: Ast -> [Instr]
30 translate ast
31   = case ast of
32       Num n -> PushConst n
33       Ident x -> PushVar x
34       Plus e1 e2 -> translate e1 ++ translate e2 ++ [Add]
```

Comments on the code;

- Note that the structures for `Token` and `Ast`, defined on lines 2 and 4 respectively, are very similar - however, the latter represents a tree structure
- We require a parsing function for each non-terminal in the code, hence we have `parseExpr` and `parseFact`
- It's easier to start with the non-recursive cases, which are the factors
- From here, you can see that the recursion structure of the code closely follows the recursion structure of the grammar, as we have the recursion in line 20 (on expressions, after the factor is parsed)
- Each function returns the part of the AST it has generated and the **remaining** tokens after consuming input
- The final translation function generates instructions for a very simple stack machine

13th January 2020

Bootstrapping

Imagine the scenario where there is a new language, and only one machine. The process of writing a compiler for this language, with this language, is to first manually write a compiler in the assembly language for the machine for a small subset of the new language. Using the subset of this new language (that can be compiled), we can write a compiler to compile more of the language, and this process continues until we can compile the entire language.

Lexical Analysis

The lexical analyser (sometimes called a scanner) converts characters into tokens. This is because the compiler shouldn't have to deal with strings directly. Normally, this removes whitespace, as it isn't needed in code generation (other than for string / character literals). In this course, the regular expressions that the scanners use will be converted into finite automata.

Identifiers are usually classified into the following;

- **keywords**

  These are defined by the language, and are reserved. For example, words such as "return", "for", "class", and so on are represented as their own tokens (`RETURN`, `FOR`, and `CLASS` respectively), since there is a (relatively) small finite set to work with. The scanner needs to be able to quickly verify if something is a keyword, and therefore something such as a "perfect" hash function is used.

- **user-defined**

  These are defined by the programmer. Since there can be (theoretically) an infinite amount of them, it's not possible to generate a unique token for each one, and therefore it usually falls under a general identifier token with a string parameter (such that "xyz" becomes `IDENT("xyz")`, or similar).

In the case of literals, some special consideration may be needed for cases where the language we are compiling can support more than the language we are writing the compiler in. For example, if the language we are writing a compiler for can support arbitrarily large integers, special consideration will be required if the language the compiler is written in cannot support such values. Some examples of literals are as follows (more can exist, such as booleans, characters, and so on);

- **integers**

  An integer would likely be represented by a general integer token, such that the string "123" would become `INTEGER(123)`, or similar.

- **strings**

  Similar to integers, but the token constructor will now take a string parameter instead of an integer, such that ""foo"" would become `STRING("foo")`.

There are other tokens, not just the two cases above, such as (but not limited to);

- **operators / symbols**

  Normally operators / symbols such as "+", "¡=", "(" are represented as their own unique token, such as `PLUS`, `LTE`, `LPAREN`, respectively.

- **whitespace / comments**

  Whitespace characters are normally removed (unless they are in the case where they exist within a literal), but are needed to separate adjacent identifiers. Comments are also usually removed.

Regular Expressions (Regex)

This allows us to formally define the acceptable tokens of the language.

regex	matches
`a`	a literal **symbol** of the language's alphabet (that isn't a regex meta-character)
ϵ	the empty string **epsilon**
`R1 R2`	**concatenation** of regex `R1` followed by `R2` (medium precedence)
`R1|R2`	**alternation** of regex `R1` or `R2` (lowest precedence)
`R*`	**repetition** of regex `R` (0 or more times) (highest precedence)
`(R)`	**grouping** `R` by itself, used to override precedence
`\a`	"escaping", used to have a literal of a meta-character
shortcut	(can be made by the rules above)
`R?`	0 or 1 occurrences of regex `R`
`R+`	1 or more occurrences of regex `R`
`[aeiou123]`	any character from the given set
`[a-zA-Z0-9]`	any alphanumeric character
`[^a-zA-Z]`	any character **except** the ones in the set
`.`	any character except a newline

15th January 2020

Regular Expression Rules

We write rules or productions in the form $\alpha \rightarrow$ `X`, where α is a non-terminal (the name of the rule), and `X` is some regular expressions constructed by any combination of terminals (symbols) and non-terminals (names of **other** rules - recursion is not allowed, therefore all non-terminals must be defined before being used in another rule). For example, we have the following regular expressions for a simple grammar (note that it looks very similar to WACC).

```
     Digit → [0-9]
       Int → Digit+
SignedDigit → (+ | -)?  Int
   Keyword = if | while | do
Identifier = Letter (Letter | Digit)*
```

However, we can run into the issue of ambiguity, when a character sequence can match to more than one regex. For example, the input string `dough` matches to the identifier `dough`, as well as partially to the keyword `do`. Two strategies are either to match the longest character sequence (causing the former), or to have textual precedence, where the first regex takes precedence (causing the latter).

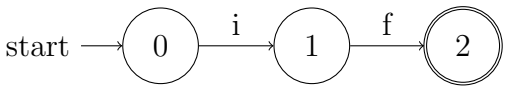
Finite Automata

When we draw a finite automata, its important to note the following symbols we use;

- **states** are circles
  - the **start** state has an unlabelled arrow going into it
  - the **accepting** (end) state is a double circle
  - all non-accepting states have arrows leading to an error state, but this is often omitted
- **transitions** are arrows between states, with the matched **symbol** being the labels of the arrows

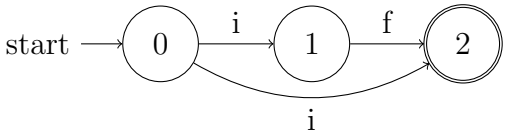
The types of finite automata we look at are the following;

- **deterministic finite automata (DFA)**



The example above is deterministic as there are no two transitions from the same state with the same symbol.

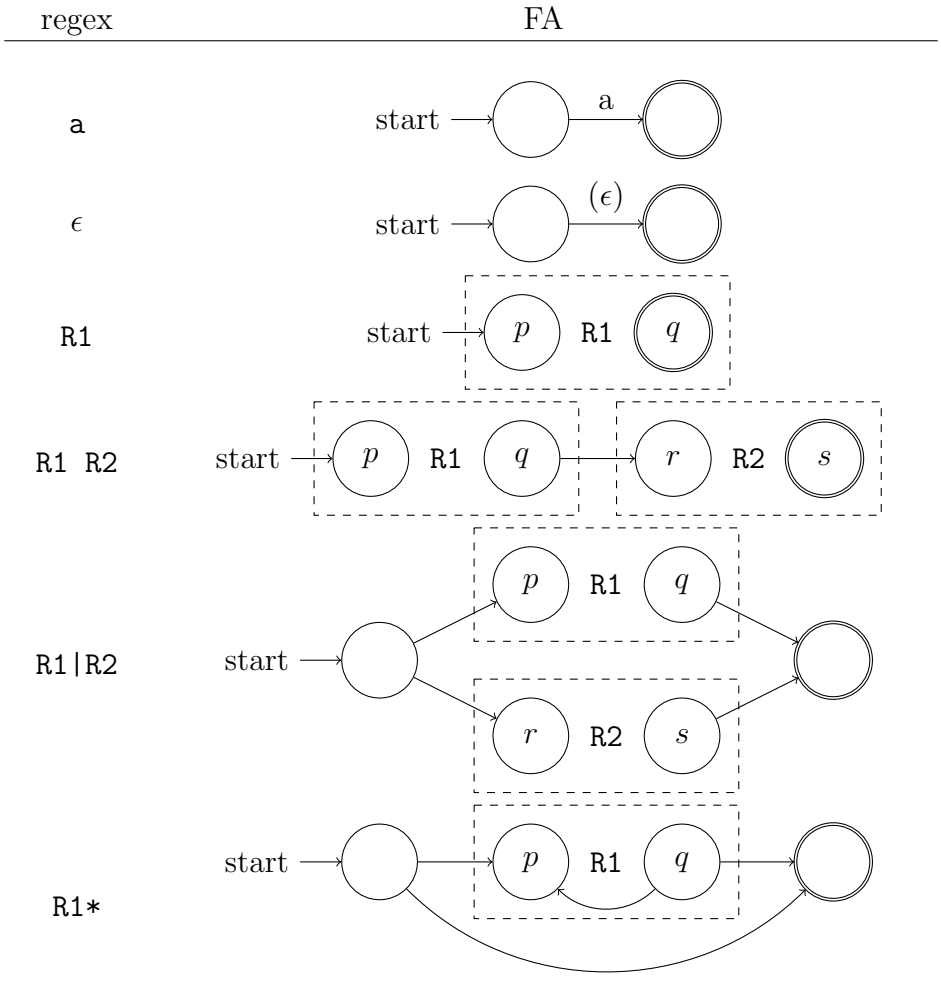
- **non-deterministic finite automata (NFA)**



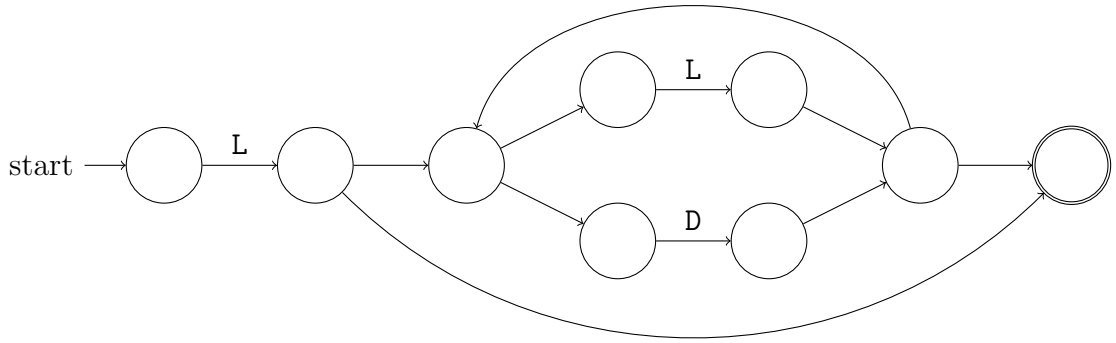
The example above is non-deterministic as there is more than one transition from state 0 with the same symbol. It allows for choice and more compact solutions, but requires backtracking.

Conversion of Regex to (Non-Deterministic) Finite Automata

Thompson's construction uses ϵ-transitions to "glue" together automata. While this is usually represented with an ϵ over the transition, it can be omitted for brevity, and therefore any transitions without labels will be assumed to be ϵ-transitions. In lieu of repeatedly drawing the same thing, let the regular expression `R1` have an initial state of p and an accepting / end state of q, and let `R2` have an initial state r and accepting state s. Also note that within the dotted lines can exist an arbitrarily complex automata.



For example, consider the regular expressions for an identifier, following the form `L(L | D)*`, which represents a letter followed by any combination of letters and digits. Note that we are allowed to abbreviate the use of `Letter` to `L`, for brevity, and similar for `Digit` to `D`.



Conversion from NFA to DFA

It's important to note the worst case complexities for NFA and DFA are as follows, where n is the length of the input string, and r is the length of the regular expression;

type	space complexity	time complexity
NFA	$O(r)$	$O(nr)$
DFA	$O(2^r)$	$O(n)$

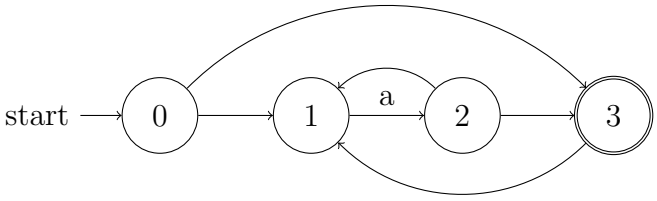
As you can see, DFAs are much faster, but can be exponentially bigger than NFAs. However, the worst case space complexity for a DFA is rarely reached for lexer analyser generators.

In order to convert from NFA to DFA, we use ϵ-closures. To avoid repetition, I will be denoting the ϵ closure of s as $\epsilon_c(s)$.

$$\epsilon_c(s) = \text{set of states reachable by zero or more } \epsilon\text{-transitions from } s$$

$$\epsilon_c(\{s_1, \ldots, s_n\}) = \bigcup_{i=1}^{n} s_i$$

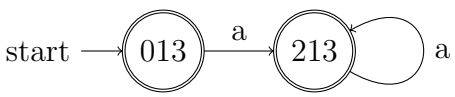
For example, take the following NFA;



Which has the following closures;

$\epsilon_c(0) = \{0, 1, 3\}$
$\epsilon_c(1) = \{1\}$
$\epsilon_c(2) = \{1, 2, 3\}$
$\epsilon_c(3) = \{1, 3\}$

From our start state, we then create a node consisting of its ϵ-closure. We look at this new node, find the ones that have a non-ϵ-transition, and handle those cases. For each new case, we take the state that it goes to, and create a node consisting of its ϵ-closure, and repeat the process. Finally, we mark each of the states that contain an accepting state as an accepting state. The example above becomes the following;



17th January 2020

At the start of this lecture, he goes over the grammar for an example language that has statements, consisting of assignment and iteration.

Code Generation for a Stack Machine

He then continues with a representation of a stack machine, it's a lot of reading, and most of it is on the slides. The general premise of the stack machine is that it fulfils a promise where it leaves the result of a computation at the top of the stack, and doesn't modify anything that was below it. As such, we end up with the following code (note that `label1` and `label2` are unique labels that haven't been used before - generating this in Haskell isn't as trivial as in other languages);

```
type Name = [Char]
type Label = [Char]

data Stat = Assign Name Exp | Seq Stat Stat | ForLoop Name Exp Exp Stat
data Exp = Binop Op Exp Exp | Unop Op Exp | Ident Name | Const Int
data Op = Plus | Minus | Times | Divide
data Instruction = Add | Sub | Mul | Div | Negate
| PushImm Int | PushAbs Name | Pop Name
| CompEq | JTrue Label | JFalse Label
| Define Label -- not executed

-- naive code generator
transStat :: Stat -> [Instruction]
transStat s
= case s of
    Assign id exp -> transExp exp ++ [Pop id]
    Seq s1 s2 -> transStat s1 ++ transStat s2
    ForLoop id e1 e2 body ->
        transExp e1 ++ [Pop id] ++ -- initialisation
        [Define label1] ++
        transExp e2 ++ [PushAbs id] ++ [CompGt] ++ [JTrue label2] ++ -- test
        transStat body ++ -- loop body
        [PushAbs id] ++ [PushImm 1] ++ [Add] ++ [Pop id] ++ -- increment
        [Jump label1] ++ -- go back to test
        [Define label2] -- define end of loop

transExp :: Exp -> [Instruction]
transExp e
= case e of
    Ident id -> [PushAbs id]
    Const v -> [PushImm v]
    Binop op e1 e2 -> transExp e1 ++ transExp e2 ++ transOp op
    Unop op e -> transExp e ++ transUnop op

transOp :: Op -> [Instruction]
transOp Plus = [Add]
transOp Minus = [Sub]
transOp Times = [Mul]
transOp Divide = [Div]

transUnop :: Op -> [Instruction]
```

```
42 transUnop Minus = [Negate]
```

We need to remember that the compiler does **not** execute code, or evaluate the instructions, since it cannot know the value of variables which are determined at runtime. Looking at line 16, in the code above, the instruction **list** `transExp exp`, after execution, leaves the result of evaluating `exp` at the top of the stack, ready for `Pop id` to store. It's also important to remember that `Define` isn't actually executed, but used for the assembler to figure out addresses for jumps.

Code Generation for a Machine with Registers

While a stack machine isn't unrealistic, it will be much slower compared to one that has efficient use of registers. For this part, we want to concentrate on the effective use of registers for arithmetic operations. In the previous code snippet given in Haskell, we modify the instructions to use registers, as such (replacing instances where sensible, otherwise preserving them);

```
1  type Reg = Int
2
3  data Instruction = Add Reg Reg | -- and so on
4    | Load Reg Name | LoadImm Reg Int | Store Reg Name | Push Reg | Pop Reg
5    | CompEq Reg Reg | JTrue Reg Label | JFalse Reg Label
6
7  transExp :: Exp -> Reg -> [Instruction]
8  transExp e r
9    = case e of
10       Ident id -> Load r id
11       Const v -> LoadImm r v
12       Binop op e1 e2 ->
13         transExp e1 r ++
14         transExp e2 (r + 1) ++
15         [binop r (r + 1)]
16         where
17           binop = case op of
18             Plus -> Add
19 -- and so on
```

Notice the additional parameter given into `transExp`. This specifies the register in which the result of the operation should go into. The allocation of registers in this translator mirrors the "slot" that the expression would be stored in on the stack. As we mirror the stack in this sense, specifying something goes into register i allows the program to modify anything $\geq i$, but nothing below it. This is why we specify the second expression must be evaluated into $r + 1$, as we don't want to modify the result of the first.

However, one caveat of this, if we were to evaluate `(x * 4) + 3`, is that we would have to use a total of 2 registers. This is because all our operations currently only work between registers, and therefore immediate values have to be loaded in. However, many assembly languages allow for immediate operations on registers, meaning that the entire execution can be achieved with only a single register. The general idea of this is that we are able to take advantage of pattern matching, given that the language the compiler is written in supports it, to look for these obvious patterns, such as doing arithmetic with a constant term. This is called **instruction selection**.

Combination of Register and Stack

When there aren't enough registers for us to use, in the case of a complex arithmetic operation for example, we want to start utilising the stack. This gives us the performance benefits of using registers, but also allows us to perform arbitrarily complex computations.

To do this, we consider the example of an accumulator machine. This only has one register and also uses the stack. Generating code for this allows us to generate code for the case where the registers are all full, except for one, which we then treat as the accumulator. Therefore the general strategy is to consider it as a register machine until all but one register is used, and then treat it as an accumulator machine.

Note that when we do work on an accumulator machine, the binary operation $e_1 \bullet e_2$ has the following order; evaluate e_2 into the accumulator, push it to the stack, and then evaluate e_1 into the accumulator, then perform an add instruction that uses both the register and the stack.

```
1  transExp :: Exp -> Reg -> [Instruction]
2  transExp e r
3    = case e of
4        -- etc
5        Binop op e1 e2 ->
6          if (r == MAXREG) then
7            transExp e2 r ++
8            [Push r] ++
9            transExp e1 r ++
10           transBinopStack op r -- one operand is register
11         else
12           transExp e1 r ++
13           transExp e2 (r + 1) ++
14           transBinop op r (r + 1) -- both operands are registers
15       -- etc
```