

# CO221 - Compilers

6th January 2020

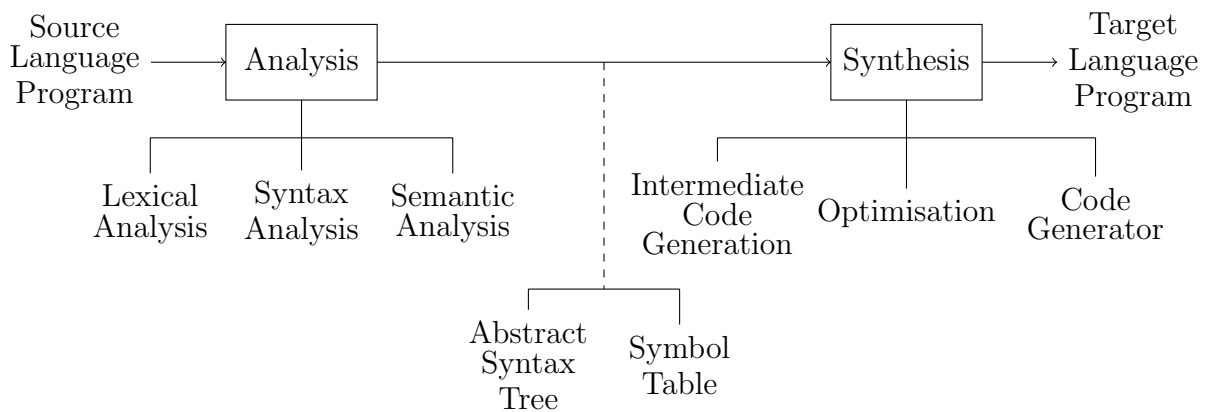
A compiler is a program which processes programs, including translating a program written in one language (usually higher level) to another programming language (usually in a lower level). In our course, the focus is to generate assembly code from the high level language. This translation goes between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

## Structure

The general structure of a compiler is as follows;

- |                     |   |
|---------------------|---|
| 1. <b>input</b>     | takes in an input program in some language                    |
| 2. <b>analysis</b>  | constructs an internal representation of the source structure |
| 3. <b>synthesis</b> | walks the representation to generate the output code          |
| 4. <b>output</b>    | 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;

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

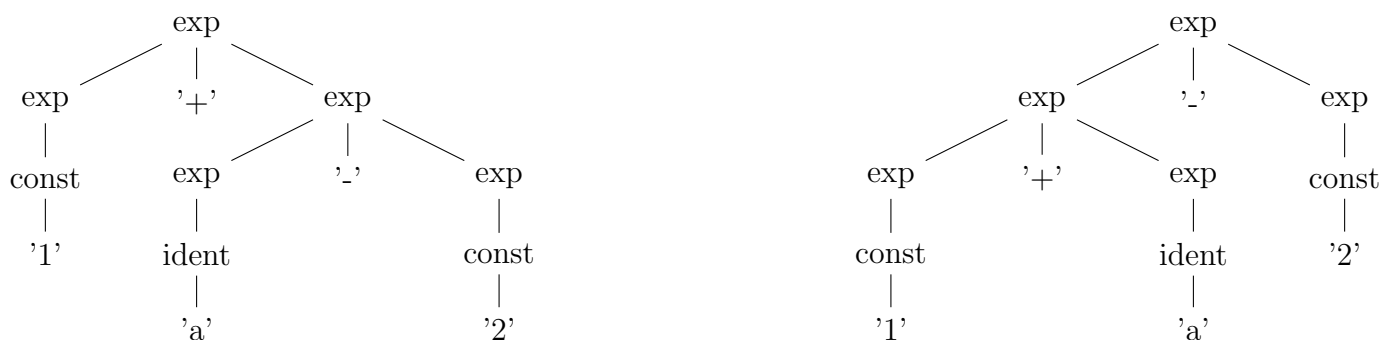
A string of only terminals (**sentential form**) can be derived using the grammar by beginning with the start symbol, and repeatedly replacing each non-terminal with the RHS from a corresponding production. We refer to the set of all sentential forms derived from the start symbol as the **language**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

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


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;

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

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

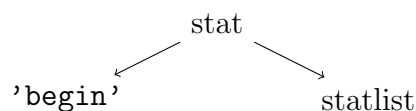
The parser checks that the input is grammatically correct, and builds an AST representing the structure. In general, there are two classes of parsing algorithms;

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

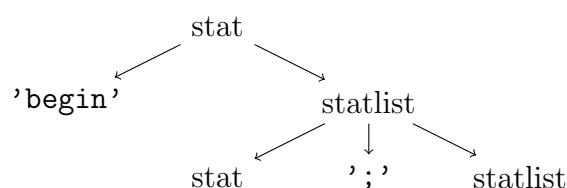
For this, we will use the input "begin S; S; end", with the following grammar;

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

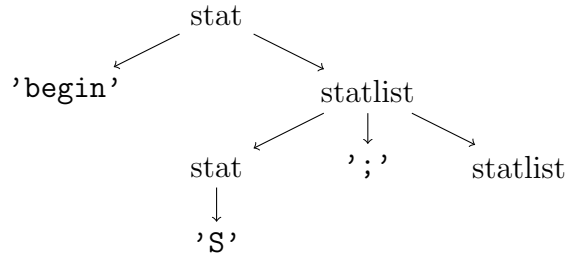
When we start top-down parsing, we start with the non-terminal stat. The first token we identify is the 'begin', thus our tree becomes the following;



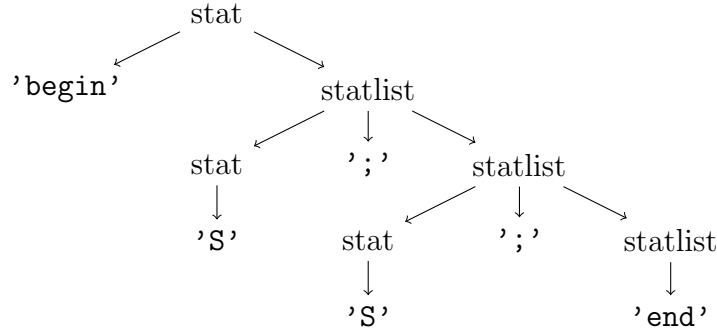
However, as the next symbol isn't the terminal 'end', we have to use an alternative. As we only have one alternative, we can predict it, and thus the tree becomes;



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	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 ';' stat ';'	statlist			
'begin' stat ';' 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;

$\text{expr} \rightarrow \text{fact '+' expr} \mid \text{fact}$   
 $\text{fact} \rightarrow \text{number} \mid \text{identifier}$

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

15th January 2020

### Regular Expression Rules

We write rules or productions in the form  $\alpha \rightarrow X$ , where  $\alpha$  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

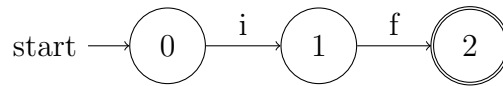
When we draw a finite automata, its important to note the following symbols we use;

- **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)**



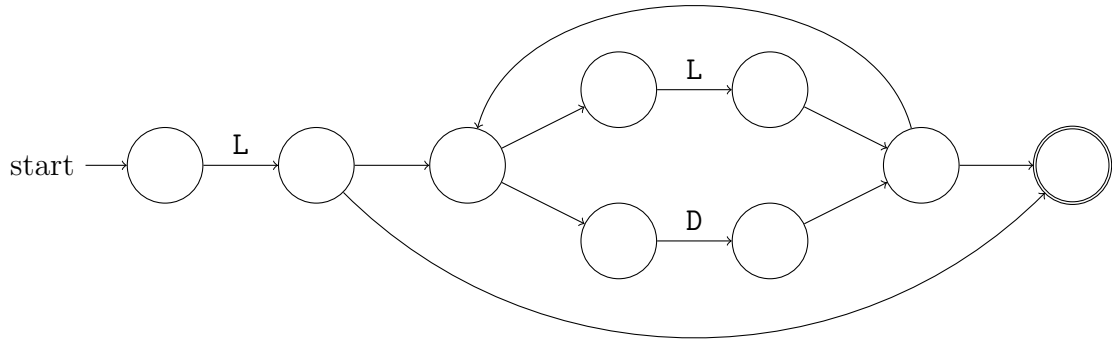
The example above is non-deterministic as there is more than one transition from state 0 with the same symbol. It allows for choice and more compact solutions, but requires backtracking.

### Conversion of Regex to (Non-Deterministic) Finite Automata

**Thompson's construction** uses  $\epsilon$ -transitions to "glue" together automata. While this is usually represented with an  $\epsilon$  over the transition, it can be omitted for brevity, and therefore any transitions without labels will be assumed to be  $\epsilon$ -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.

regex	FA
<b>a</b>	<pre> graph LR     start((start)) -- a --&gt; accept((( )))   </pre>
$\epsilon$	<pre> graph LR     start((start)) -- "(\epsilon)" --&gt; accept((( )))   </pre>
<b>R1</b>	<pre> graph LR     start((start)) --&gt; p((p))     subgraph R1 [R1]         p --&gt; q(((q)))     end   </pre>
<b>R1 R2</b>	<pre> graph LR     start((start)) --&gt; p((p))     subgraph R1 [R1]         p --&gt; q(((q)))     end     q --&gt; r((r))     subgraph R2 [R2]         r --&gt; s(((s)))     end   </pre>
<b>R1   R2</b>	<pre> graph LR     start((start)) --&gt; p((p))     start --&gt; r((r))     subgraph R1 [R1]         p --&gt; q(((q)))     end     subgraph R2 [R2]         r --&gt; s(((s)))     end   </pre>
<b>R1*</b>	<pre> graph LR     start((start)) --&gt; p((p))     subgraph R1 [R1]         p --&gt; q(((q)))         q --&gt; p     end   </pre>

For example, consider the regular expressions for an identifier, following the form  $L(L \mid D)^*$ , 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	$O(2^r)$	$O(n)$

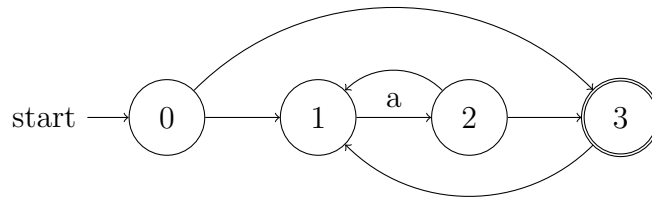
As you can see, DFAs are much faster, but can be exponentially bigger than NFAs. However, the worst case space complexity for a DFA is rarely reached for lexer analyser generators.

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

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

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

For example, take the following NFA;



Which has the following closures;

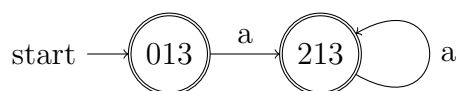
$$\epsilon_c(0) = \{0, 1, 3\}$$

$$\epsilon_c(1) = \{1\}$$

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

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

From our start state, using subset construction, we then create a node consisting of its  $\epsilon$ -closure. We look at this new node, find the ones that have a non- $\epsilon$ -transition, and handle those cases. For each new case, we take the state that it goes to, and create a node consisting of its  $\epsilon$ -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);

```
1  type Name = [Char]
2  type Label = [Char]
3
4  data Stat = Assign Name Exp | Seq Stat Stat | ForLoop Name Exp Exp Stat
5  data Exp = Binop Op Exp Exp | Unop Op Exp | Ident Name | Const Int
6  data Op = Plus | Minus | Times | Divide
7  data Instruction = Add | Sub | Mul | Div | Negate
8  | PushImm Int | PushAbs Name | Pop Name
9  | CompEq | JTrue Label | JFalse Label
10 | Define Label -- not executed
11
12 -- naive code generator
13 transStat :: Stat -> [Instruction]
14 transStat s
15 = case s of
16     Assign id exp -> transExp exp ++ [Pop id]
17     Seq s1 s2 -> transStat s1 ++ transStat s2
18     ForLoop id e1 e2 body ->
19         transExp e1 ++ [Pop id] ++ -- initialisation
20         [Define label1] ++
21         transExp e2 ++ [PushAbs id] ++ [CompGt] ++ [JTrue label2] ++ -- test
22         transStat body ++ -- loop body
23         [PushAbs id] ++ [PushImm 1] ++ [Add] ++ [Pop id] ++ -- increment
24         [Jump label1] ++ -- go back to test
25         [Define label2] -- define end of loop
26
27 transExp :: Exp -> [Instruction]
28 transExp e
29 = case e of
30     Ident id -> [PushAbs id]
31     Const v -> [PushImm v]
32     Binop op e1 e2 -> transExp e1 ++ transExp e2 ++ transOp op
33     Unop op e -> transExp e ++ transUnop op
34
35 transOp :: Op -> [Instruction]
36 transOp Plus = [Add]
37 transOp Minus = [Sub]
38 transOp Times = [Mul]
39 transOp Divide = [Div]
40
41 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.

```

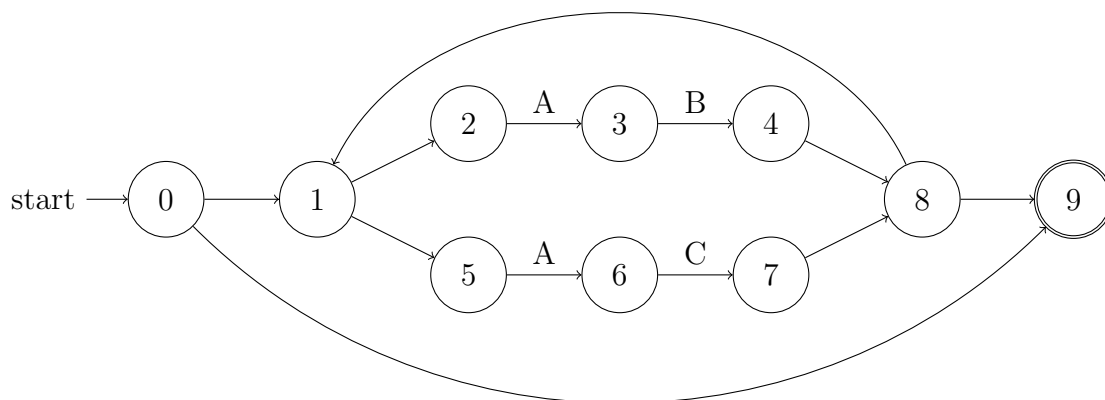
1  transExp :: Exp -> Reg -> [Instruction]
2  transExp e r
3    = case e of
4        -- etc
5        Binop op e1 e2 ->
6          if (r == MAXREG) then
7            transExp e2 r ++
8            [Push r] ++
9            transExp e1 r ++
10           transBinopStack op r -- one operand is register
11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

```

**20th January 2020**

### Subset Construction of Regex

This lecture starts by working through the subset construction of the regex  $(AB|AC)^*$ . We start by constructing the NFA for the corresponding regular expression;

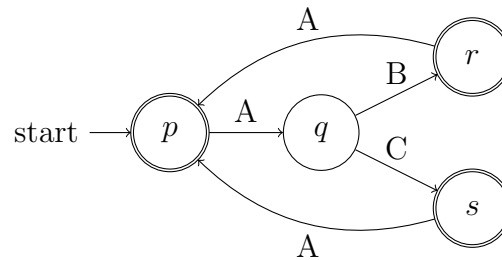


The DFA is then generated as follows (note that in lieu of writing out the entire closure, I use a single character, however it's best to write out the entire thing);

1. create a state for 0 with its closure (01259), let it be  $p$
2. looking at the states within  $p$ , we see that the ones with non- $\epsilon$ -transitions are 2 and 5, both with transitions with label 'A', to 3 and 6 respectively
3. create a state for 3 and 6, with its closure (36), let it be  $q$

4. looking at the states within  $q$ , we see that the ones with non- $\epsilon$ -transitions are both 3 and 6, with transitions 'B' to 4, and 'C' to 7 respectively
5. create a state for 4, with its closure (124589), let it be  $r$
6. looking at the states within  $r$ , we see that the ones with non- $\epsilon$ -transitions are 2 and 5 - however we have already seen this, it goes to the state  $q$
7. create a state for 7, with its closure (125789), let it be  $s$
8.  $s$  also goes to  $q$  for the same reasoning as above
9. all the states that contain 9 are now marked as accepting states, which are  $p, r, s$

This gives us the following DFA



## LR (Bottom-up) Parsing

The rules in the context free grammar exist as  $R \rightarrow (R|t)^*$ , where  $R$  is a rule, and  $t$  is a token. Therefore rules are represented as some arbitrary combination of rules and tokens (thus supporting recursion, unlike regular expressions). The goal of a parser is to convert a sequence of tokens into an AST (or parse tree). In the case of the AST we can discard tokens that are simply syntactic sugar, as the structure of the program is represented in the tree.

Note that we need to look at subsets of CFGs, as they can be cubic ( $O(n^3)$ ) to parse in the worst case, and we want to consider the largest subsets which take  $O(n)$ . The types we consider are as follows (note that  $LL(n) \subseteq LR(n)$ , therefore LR parsers are more powerful);

- $LL(k)$  left to right scanning, with  $k$  token look-ahead, and left-most derivation top-down, as it builds the AST from the root nodes to the leaf nodes
- $LR(k)$  left to right scanning, with  $k$  token look-ahead, and right-most derivation bottom-up, as it builds the AST from the leaf nodes to the root node

We will be first looking at  $LR(0)$ , then  $LR(1)$ , and then finally  $LALR(1)$ .

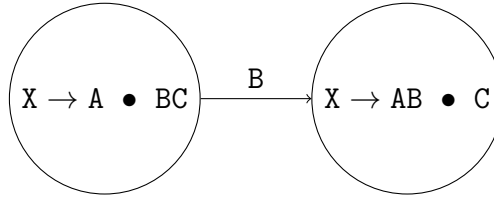
Starting with  $LR(0)$ , it doesn't need the token for a reduce. An  $LR(0)$  item is a "rule" with a dot ( $\bullet$ ) at some position in the right hand side. An item indicates how much of a rule we've seen. For example, the item  $E \rightarrow E + \bullet \text{ int}$  indicates that we've seen an expression, followed by a plus, and we are hoping to see an integer to complete the item. This means that  $LR(0)$  items represent the steps to recognise the RHS of a given production.

For example, we can look at the rules for  $X \rightarrow ABC$ ;

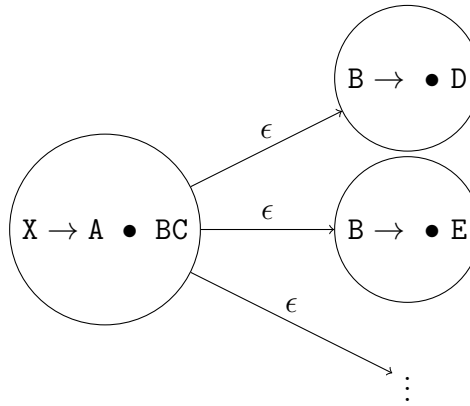
$X \rightarrow \bullet ABC$	initial item
$X \rightarrow A \bullet BC$	
$X \rightarrow AB \bullet C$	
$X \rightarrow ABC \bullet$	complete item

$$X \rightarrow \underbrace{AB}_{\text{seen}} \bullet \overbrace{C}^{\text{hope to see}}$$

These items are used as states of a finite automaton to maintain information about the progress of a shift-reduce parser. We can build an NFA from these items as follows (and then build a DFA with subset construction). Working with the single rule  $X \rightarrow A \bullet BC$ , we add the following transition;



Additionally, if B is non-terminal, such that  $B \rightarrow D \mid E \mid \dots$ , add an  $\epsilon$ -transition for each rule;



We also need to add a start rule, with symbol to indicate the end of input \$ (it is implied if omitted). For example, given the grammar

$$E \rightarrow E \text{ ' + ' } \text{int} \mid \text{int}$$

We'd add a rule  $E' \rightarrow E \$$ . This therefore has a total of 8 items;

$E' \rightarrow \bullet E$	initial item
$E' \rightarrow E \bullet$	complete / reduce item
$E \rightarrow \bullet E + \text{int}$	initial item
$E \rightarrow E \bullet + \text{int}$	
$E \rightarrow E + \bullet \text{int}$	
$E \rightarrow E + \text{int} \bullet$	complete item
$E \rightarrow \bullet \text{int}$	initial item
$E \rightarrow \text{int} \bullet$	complete item

## Chomsky Hierarchy

In the following, R is a non-terminal (name of a rule),  $\mathbf{t}$  is a sequence of terminals, and  $\alpha, \beta, \varphi$  are sequences of terminals and non-terminals.

type 3: $R \rightarrow \mathbf{t}$	regular grammars (DFA)
type 2: $R \rightarrow \alpha$	context free grammars (pushdown automata)
type 1: $\alpha R \beta \rightarrow \alpha \varphi \beta$	context sensitive grammars (linear bounded automata)
type 0: $\alpha \rightarrow \beta$	unrestricted grammars

22nd January 2020

## DFA to LR(0) Parsing Table

From the LR(0) items, it's fairly straightforward to determine the  $\epsilon$ -closures, therefore the DFA can be constructed directly (however it can also be constructed with subset construction from a NFA). Continuing on with the simple expression example from last lecture, we can construct the DFA with the following rules in each state;

state 0:

$E' \rightarrow \bullet E$   
 $E \rightarrow \bullet E + \text{int}$   
 $E \rightarrow \bullet \text{int}$

state 1:

$E' \rightarrow E \bullet$   
 $E \rightarrow E \bullet + \text{int}$

state 2:

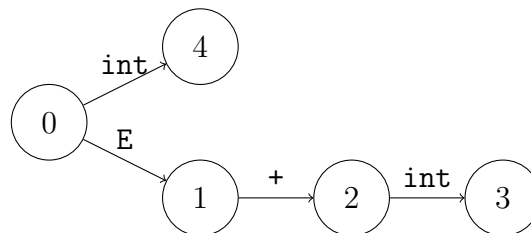
$E \rightarrow E + \bullet \text{int}$

state 3:

$E \rightarrow E + \text{int} \bullet$

state 4:

$E \rightarrow \text{int} \bullet$



Converting from this to a parsing table, we follow these rules;

- for each terminal transition  $X \xrightarrow{T} Y$  add  $P[X, T] = sY$  (shift Y)  
 "move from state X to state Y"
- for each non-terminal transition  $X \xrightarrow{N} Y$  add  $P[X, N] = gY$  (goto Y)
- for each state X containing  $R' \rightarrow \dots \bullet$  add  $P[X, \$] = a$  (accept)  
 this corresponds to the auxiliary rule, when we are done parsing
- for each state X which contains a  $R \rightarrow \dots \bullet$  add  $P[X, T] = rn$  (reduce) for every terminal T, where n is R's rule number

the number does not correspond to a state, it instead corresponds to a rule number, by looking at the corresponding rule, we know how many items need to be popped off the stack and reduced

For the DFA above, we can represent it as the following LR(0) parsing table; Anything left blank is considered an error.

state	action			goto
	int	+	\$	E
0	s4			g1
1		s2	a	
2	s3			
4	r1	r1	r1	
3	r2	r2	r2	



## Model of an LR Parser

Assuming all the states are integers, we can approach the LR parser as follows;

1. push state 0 onto the stack
2. repeatedly decide what to do next depending on the top of the stack and the current character (`switch P[S[top], curr]`)
  - shift  $n$  push the state  $n$  onto the stack, and go to the next token
  - reduce  $n$  (complex)
    - remove  $K$  elements off the stack, where  $K$  is the length of the RHS of **rule  $n$**
    - push  $P[S[top], L]$  where  $L$  is the LHS of **rule  $n$** ; this essentially looks at the state it was in before, and performs the "goto"  
in our example if we reduced to an  $E$ , state 0 is most likely on the top of the stack, and therefore it goes to state 1
    - generate an AST node with this data for the rule
  - accept
  - error report an error
  - goto  $n$  not directly selected, looked up in reduce case

## FIRST and FOLLOW Sets

While it is possible to formally write out the algorithm for deriving the sets, it's easier to do it by intuition.

### • FIRST set

The FIRST set of a terminal is itself. However, the FIRST set for a rule is anything that can begin the derivation for such that particular rule. This set is constructed recursively, by looking through the FIRST sets of the first item of the LHS of each rule. However, note that because we are generating a set, we can stop when we encounter something we've already checked, as it wouldn't be added anyways. Let the rule be  $R$ , let  $X$  be an arbitrary rule, and let  $t$  be a terminal;

- $R \rightarrow t \dots$  starts with a terminal  
add  $t$  to the FIRST set of  $R$
- $R \rightarrow X \dots$  starts with a non-terminal  
add the FIRST set of  $X$ , this can lead to recursion (which can be dealt with)

### • FOLLOW set

The FOLLOW set of a rule is something that can come after it. The intuition in this is to look at how it is used in the RHS of productions. On the RHS, there are three cases (or more, I could be wrong), let the rule be  $R$ , let  $X, Y$  be arbitrary rules, and let  $t$  be a terminal;

- $X \rightarrow \dots R$  at the end of a rule  
add the FOLLOW set of  $X$ , as anything following  $X$  could follow  $R$
- $X \rightarrow \dots R t \dots$  followed by a terminal  
add  $t$  to the FOLLOW set of  $R$
- $X \rightarrow \dots R Y \dots$  followed by a non-terminal / rule  
add the FIRST set of  $Y$ , as anything that could start a derivation of  $Y$  could follow  $X$

Note that the same intuition on recursion applies. For example, in the first case, if  $R = X$ , then the FOLLOW set of  $R$  needs the FOLLOW set of  $R$ , but there's no point in doing this.

It's also important to consider the case where it can be  $\epsilon$ .

## Weights

In the grammar, we've defined expressions to be right associative, hence it is a right growing tree. This requires more registers if we evaluate the left side first, as we need to retain this result for the next calculation. This can be shown in the example below, for the evaluation of  $1 + 2 + 3$ , with the left tree being left associative  $(1 + 2) + 3$ , and the right tree being right associative  $1 + (2 + 3)$  - note that the former uses 2 registers, whereas the latter uses 3.



The general idea for this is to first evaluate the subexpression that requires more registers. For example, let there be a binary operator with operands  $e_1$  and  $e_2$ , needing  $L$  registers, and  $R$  registers respectively. If we choose to evaluate  $e_1$  first, then we require  $L$  registers, and  $R + 1$  registers - this is because in the evaluation of  $e_2$ , we need to use  $R$  registers, as well as maintaining the result of  $e_1$ . Therefore, the cost (in registers) for evaluating  $e_1$  first is  $c_1 = \max\{L, R + 1\}$ . Under similar reasoning, the cost of evaluating  $e_2$  first is  $c_2 = \max\{L + 1, R\}$ . Therefore, assuming we always choose the optimal subtree in terms of registers, we have the overall cost of evaluating the binary operation is  $\min\{c_1, c_2\}$ . We can represent the weights as follows;

```

1 weight :: Exp -> Int
2 weight (Const _) = 1
3 weight (Ident _) = 1
4 weight (Binop _ e1 e2) = min c1 c2
5   where
6     c1 = max (weight e1) (weight e2) + 1
7     c2 = max (weight e1) + 1 (weight e2)

```

However, when we specify the target registers (naively) we have one of two issues;

- it stores the result of  $e_2$  into  $r$ , and  $e_1$  into  $(r + 1)$ , when  $e_2$  is evaluated first, this leads to issues with operators that aren't commutative
- trying to fix the above, and storing  $e_2$  into  $(r + 1)$ , then evaluating  $e_1$  into  $r$  - the evaluation of  $e_1$  may modify the contents of  $(r + 1)$ , hence overwriting the result of  $e_2$

The fix is to give the expression translation function a list of registers it's able to use. The result should be stored into the first register in that list.

```

1 transExp :: Exp -> [Register] -> [Instruction]
2 transExp (Const n) (dst:_) = [LoadImm dst n]
3 transExp (Ident x) (dst:_) = [LoadAbs dst x]
4 transExp (Binop op e1 e2) (dst:nxt:rest)
5   | weight e1 > weight e2 =
6     transExp e1 (dst:nxt:rest) ++
7     transExp e2 (nxt:rest) ++
8     transBinop op dst nxt
9   | otherwise =
10    transExp e1 (dst:nxt:rest) ++
11    transExp e2 (nxt:rest) ++
12    transBinop op dst nxt

```

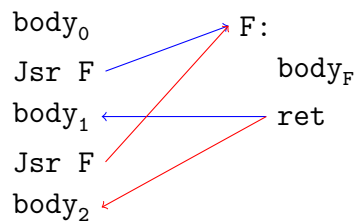
However, this can be further optimised by using more advanced instructions such as doing immediate addressing whenever possible. The translator can use pattern-matching to detect these cases.

The worst case for this is a perfectly balanced tree. This is because if it is unbalanced, then we can always choose a more optimal side. When there are  $k$  operators, and  $k - 1$  intermediate values, the number of registers is the depth of the tree ( $\lceil \log_2(k) \rceil$ ). This means that with  $N$  registers, we can support expressions with up to  $2^N$  operators.

Note that the algorithm above can lead to issues when working with side effecting expressions, as the order of execution is important. For example,  $(x)+(x++)$  is different from  $(x++)+(x)$ .

## Function Calls

At the function call, registers may already be in use, and therefore those will need to be saved. Note that the address of the next instruction also has to be saved onto the stack, for the function to return to the correct place (notice how the control paths are coloured). Consider the example below, where  $\text{body}_0$  requires the set of registers  $M$ ,  $\text{body}_1$  requires set  $N$ , and  $\text{body}_F$  requires set  $P$ .



There are two conventions (and a hybrid of the two) for saving registers with regards to calls;

- **caller-saved**

Before the instruction `Jsr F` is executed (after  $\text{body}_0$ ), it saves the intersection of registers currently in use, and registers that the function will use ( $M \cap P$ ). After the function returns, it restores ( $M \cap P$ ).

- **callee-saved**

Since the callee doesn't know which jump it's coming from, it has to save everything that **might** be in use. Before  $\text{body}_F$  is executed, it saves the set of registers  $(M \cup N) \cap P$ , and then restores it before returning.

## Optimising Compilers

While the Sethi-Ullman algorithm we followed is fast, and very easy to test, it's essentially a tree walk. Therefore it doesn't take into context what it's translating (other than the small optimisation we used to determine weights). Currently, in our implementation, we don't use registers to carry values between states, and only use them to store intermediate values for computation. An optimising compiler can have named variables in the register, with no reference to the main memory, therefore being much faster. Note that Sethi-Ullman is optimal in trees that have no shared subtrees.

## Graph Colouring

An obvious optimisation of the code below;

```

1  a1 := b1 + s * k
2  a2 := b2 + s * k
  
```

would be compute the value of  $s * k$  into a temporary variable (stored in a register) and use it for both expressions;

```

1  t  := s * k
2  a1 := b1 + t
3  a2 := b2 + t

```

We need to consider all variables on equal terms (not just ones defined by the programmer) - including intermediate values during computation.

This can be achieved by doing the following;

- (1) perform a simple tree walk to generate intermediate code where temporary values are saved in named locations (three address code - looks similar to assembly but with infinite registers)
- (2) construct an interference graph, where the nodes are temporary locations, and an arc between two nodes represents an overlap in live ranges (if they must be stored simultaneously)
- (3) colour the graph, with each register being its own colour - no connected nodes can have the same register

This is very straightforward in the case of straight line code, as we simply consider the live-in to be the first use of the value (when it is first declared), and the live-out to be the last use. Note that the LHS and RHS of an assignment does not count as a simultaneous use.

```

1  A := e1
2  B := e2
3  ...
4  ... B ...
5  C := A + B
6  ...
7  D := A * 5
8  ... D ...
9  ... C ...

```

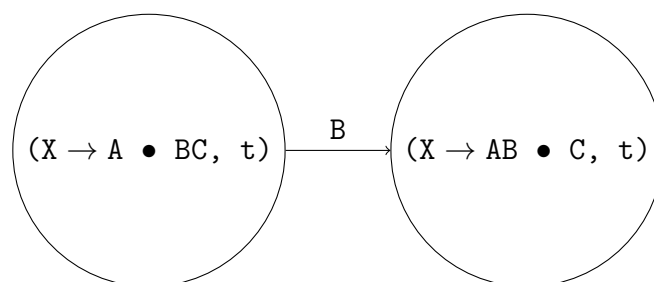
This gives the following interference graph, and the result of the colouring;



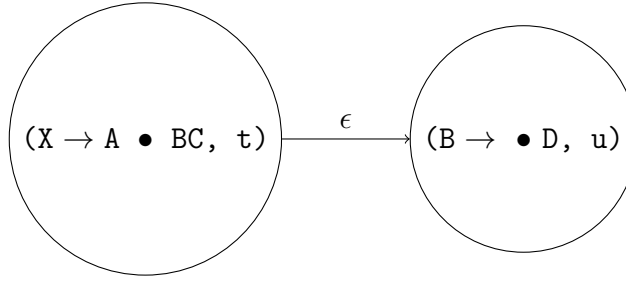
**27th January 2020**

## LR(1) Items

An LR(1) item is a pair consisting of an LR(0) item, and a lookahead token  $t$ . The process for a terminal doesn't change much;



However, if  $B$  is a rule, then for each rule in the form  $B \rightarrow D$ , we add an  $\epsilon$ -transition for every token  $u$  in  $\text{FIRST}(Ct)$ , we add a transition if and only if the string following  $B$  is derivable from  $Ct$ .



The auxiliary item  $(X' \rightarrow \bullet X, \$)$  is also added to start the construction of the parsing table. The reduction is only done if the current token matches  $t$ . Consider the following grammar (not written with the vertical separation), and the states of the DFA;

$S' \rightarrow \underline{S} \ \$$	
$S \rightarrow id$	r1
$S \rightarrow \underline{V} = \underline{E}$	r2
$V \rightarrow id$	r3
$E \rightarrow \underline{V}$	r4
$E \rightarrow int$	r5

state 0:

$S' \rightarrow \bullet \underline{S}, \$$	starting state [1]
$S \rightarrow \bullet id, \$$	derived from non-terminal in (1) [2]
$S \rightarrow \bullet \underline{V} = \underline{E}, \$$	derived from non-terminal in (1) [3]
$V \rightarrow \bullet id, =$	derived from non-terminal in (3) [4]

state 1:

$S' \rightarrow \underline{S} \bullet , \$$	shift from (1) [5]
---	--------------------

state 2:

$S \rightarrow id \bullet , \$$	shift from (2) [6]
$V \rightarrow id \bullet , =$	shift from (4) [7]

state 3:

$S \rightarrow \underline{V} \bullet = \underline{E}, \$$	shift from (3) [8]
---	--------------------

state 4:

$S \rightarrow \underline{V} = \bullet \underline{E}, \$$	shift from (8) [9]
$E \rightarrow \bullet int, \$$	derived from non-terminal in (9) [10]
$E \rightarrow \bullet \underline{V}, \$$	derived from non-terminal in (9) [11]
$V \rightarrow \bullet id, \$$	derived from non-terminal in (11) [12]

state 5:

$S \rightarrow \underline{V} = \underline{E} \bullet , \$$	shift from (8) [13]
--	---------------------

state 6:

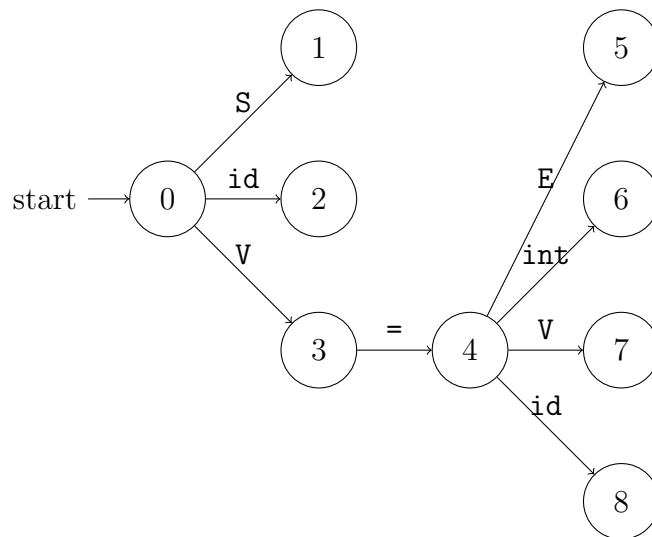
$E \rightarrow int \bullet , \$$	shift from (10) [14]
----------------------------------	----------------------

state 7:

$E \rightarrow \underline{V} \bullet , \$$	shift from (11) [15]
--	----------------------

state 8:

$V \rightarrow id \bullet , \$$	shift from (12) [16]
---------------------------------	----------------------



This is represented in the following parsing table - note that the reduce state now only happens when it matches  $\tau$ ;

state	action				goto		
	id	int	=	\$	<u>E</u>	<u>V</u>	<u>S</u>
0	s2					g3	g1
1				a			
2			r3	r1			
3			s4				
4	s8	s6			g5	g7	
5				r2			
6				r5			
7				r4			
8				r3			

## LALR(1)

In reality, LR(1) parsers don't really exist. We optimise LR(1) by merging LR(1) items that have the same LR(0) item, but different lookaheads into a single LALR(1) item that has the LR(0) item and the combined set of lookaheads. For example, we can combine these states;

[LR(1)] state 2:

$X \rightarrow \underline{E} \bullet T, =$

$V \rightarrow id \bullet , +$

[LR(1)] state 3:

$X \rightarrow \underline{E} \bullet T, \$$

$V \rightarrow id \bullet , /$

[LALR(1)] state 23:

$X \rightarrow \underline{E} \bullet T, \{=, \$\}$

$V \rightarrow id \bullet , \{+, /\}$



## Ambiguity and Conflicts

Ambiguity can be removed by rewriting the grammar, or adding additional rules. For example, we can get a shift-reduce conflict when we don't know whether to reduce an item or to continue, or we can get a reduce-reduce conflict, when it could reduce to either rule;

shift-reduce conflict

$A \rightarrow id$	can be reduced
$A \rightarrow id \text{ ' [' Expr ' ] ' }$	can be shifted

reduce-reduce conflict

$Expr \rightarrow id$   
 $Var \rightarrow id$

Similar to lexing, we'd normally continue shifting in the first case, and in the second case we give precedence to the first rule declared.

## Parse Trees

When we perform a shift, we are recognising an input and it builds a leaf node for the parse tree. On the other hand, when we perform a reduce, we take the leaf nodes, and put them into a parent node for the parse tree. However, this tree can have a lot of junk in it (thus being inconvenient to work with), and is either pruned in a separate pass to generate an abstract syntax tree (AST), or generated with instructions augmented onto the existing rules for the grammar to generate nodes as it is being parsed.

## 29th January 2020

### LL Parsing

LL parsing is a natural way of hand writing a compiler, as we don't work with DFAs, but instead work with functions for each production. This also can easily produce AST nodes at the same time. A grammar is considered  $LL(k)$  if a  $k$ -token lookahead is sufficient to determine the next step (alternative to a rule).

### Extended BNF

Due to the frequency of repeated and optional constructs in language syntax, many specifications have additional shortcuts for context free grammars;

- $\{\alpha\}$  zero or more occurrences of  $\alpha$
- $[\alpha]$  zero or one occurrences of  $\alpha$
- $(\alpha)$  grouping, useful for alternatives like  $(\alpha \mid \beta \mid \gamma)$

### Grammar to Functions

We assume there is some global variable `token` which holds the current token. It's also important to note that a rule `A` goes to a parsing function for that rule `A()`, and a token `T` goes to a match function `match(T)`, however in the code below, we will be using the function for rules (substitute `A()` for `match(A)` if we have a token instead);

```
1 "A B" => A(); B();
2 "A|B" => if token in FIRST(A): A();
3         elif token in FIRST(B): B();
```

```

4         # FIRST(A) and FIRST(B) must be disjoint
5 "{A}" => while token in FIRST(A): A();
6         # FIRST(A) must be disjoint with what follows [A]
7 "[A]" => if token in FIRST(A): A();
8         # FIRST(A) must be disjoint with what follows [A]

```

For example, when we work with the following grammar, we can create the functions as follows;

```

Statement → IfStatement | BeginStatement | PrintStatement
IfStatement → 'if' Expr 'then' Statement ['else' Statement] 'fi'
BeginStatement → 'begin' Statement {';' Statement} 'end'
PrintStatement → 'print' Expr

```

With the following code listing, for the structure of the AST, you can see it has a direct parallel with the grammar;

```

1 class StatementAST extends AST:
2     ...
3 class IfAST extends StatementAST:
4     ExprAST e
5     StatementAST t, f
6 class BeginAST extends StatementAST:
7     StatementAST[] statlist
8 class PrintAST extends StatementAST:
9     ExprAST e

```

From here, we can also directly generate the AST.

```

1 def Statement():
2     if token == IF:         return IfStatement()
3     elif token == BEGIN: return BeginStatement()
4     elif token == PRINT: return PrintStatement()
5     else: error()
6 def IfStatement():
7     match(IF)
8     e = Expr()
9     match(THEN)
10    t = Statement()
11    if token == ELSE:
12        match(ELSE)
13        f = Statement()
14    else:
15        f = None
16    match(FI)
17    return IfAST(e, t, f)
18 def BeginStatement():
19    match(BEGIN)
20    statlist = []
21    statlist.append(Statement())
22    while token == SEMICOLON:
23        match(SEMICOLON)
24        statlist.append(Statement())
25    match(END)
26    return BeginAST(statlist)
27 def PrintStatement():

```



```

28     match(PRINT)
29     return PrintAST(Expr())

```

However, it's obvious from these examples that it cannot deal with left recursion. If we had an additional rule that was in the form  $\text{Expr} \rightarrow \text{Expr} \text{ '+' } \text{Expr}$ , we'd end up recursing infinitely. Translation from a non-LL(1) grammar to a LL(1) grammar cannot be done fully automatically, and requires some ingenuity - it needs to ensure the transformed grammar has the same semantics. These three transformations are commonly used;

(1) left factorisation usually carried out after the others

if two or more alternatives share a prefix, we can factor that out, such that (in EBNF)  $A \rightarrow B \ C \mid B \ D$  can be factored out as  $A \rightarrow B \ (C \mid D)$ , and  $A \rightarrow B \mid B \ C$  would become  $A \rightarrow B \ [C]$  (these would need auxiliary rules in BNF)

(2) substitution

if there is a non-terminal B on the right hand side of a rule, it may make sense to replace B with a group consisting of the alternatives for B - this can make the grammar clearer, and also easier to do left factorisation

(3) left recursion removal

issues with left recursion have already been discussed, the idea is to write a left recursive rule with right repetition, hence  $A \rightarrow X \mid A \ Y$  should become  $A \rightarrow X \{ Y \}$ , or more generally;

$$\begin{aligned}
 A &\rightarrow X_1 \mid \dots \mid X_m \mid A \ Y_1 \mid \dots \mid A \ Y_n && \Rightarrow \\
 A &\rightarrow (X_1 \mid \dots \mid X_m) \mid A \ (Y_1 \mid \dots \mid Y_n) && \Rightarrow \\
 A &\rightarrow (X_1 \mid \dots \mid X_m) \{Y_1 \mid \dots \mid Y_n\}
 \end{aligned}$$

This is harder to deal with when the left recursion isn't direct.

## Error Recovery

Ideally, we want to be able to pick up as many (meaningful) errors as possible, and report them to the user - ensuring that the errors are accurate. To pick up multiple errors, we need a way of recovering from an error, and attempting to continue parsing.

One method of error recovery is to use the concept of a "syncset". If we don't encounter the token we were expecting, we can skip to a point where we do encounter it, or skip to the syncset of the higher level call. Generally, the syncsets grow as you go deeper into the AST, as there are more points it can attempt to "recover" to.

## 31st January 2020

### When Colouring Fails

This continues on the last lecture in this part of the course, where we ended on graph colouring. In complex programs, we are likely to run into the case where colouring fails - when we simply don't have enough registers to use for a complex graph that has too many overlapping live ranges.

One method of tackling this is to split a live range, by spilling (saving) registers that aren't used onto the stack, and then restoring them when they are needed. Ideally, this makes the graph colourable, and doesn't hurt the performance of our compiled program too much. Consider these three examples;

(1) Even though A is used after the loop, there isn't any point in keeping it in a register, especially since the loop can be intensive, and require heavy use of registers.

```

1 A = ... # 'A' declared
2 for (...) {
3     # high register pressure
4     # 'A' not used
5 }
6 ... A ... # 'A' used

```

- (2) This highlights the idea that while A isn't used in an area of high register pressure, it still makes sense to keep it live at points - hence we don't simply choose a variable to live on the stack, but instead pick points where it is saved when not needed, and recovered before it is.

```

1 A = ... # 'A' declared
2 for (...) {
3     ... A ... # 'A' used
4 }
5 for (...) {
6     # high register pressure
7     # 'A' not used
8 }
9 for (...) {
10     ... A ... # 'A' used
11 }

```

- (3) While the loop uses A within it, it isn't always needed as it is conditionally used, whereas there are other registers that would be needed unconditionally, therefore it doesn't make sense for A to be stored for the entire body of the loop.

```

1 A = ... # 'A' declared
2 for (...) {
3     # high register pressure
4     # 'A' not used
5     if (...) {
6         ... A ... # 'A' used
7     }
8 }

```

## Optimisations

It's important to note that a good optimisation needs a combination of high and low level principles. High level principles, such as inlining, are useful for picking up optimisations that are architecture-independent. Inlining replaces function calls with the body of the function, which removes the overhead required by performing the calls and returning, as well as opening up possibilities for more optimisations. For example, if we inline a function into a loop, and find loop invariant code, then that could be taken out (whereas it wouldn't have been possible with calls, as the function doesn't know where it's being called from).

On the other hand, low level principles are much more architecture dependent, such as instruction selection, and require a deeper understanding of the internals of the machine.

Peephole optimisation works at an instruction level, and is able to generate "smarter" code by removing pointless actions, for example;

```

1 peep :: [Instruction] -> [Instruction]
2 peep (Store r1 dst : Load r2 src : rest)
3   | src == dst = Store r1 dst : (peep (Load r2 r1 : rest))
4   | otherwise  = Store r1 dst : (peep (Load r2 src : rest))
5 -- etc

```

However, this also raises the question; what order should optimisations be done in? While all optimisations (hopefully) improve the code, it can also make it more complex - leading to difficulties when trying to apply another optimisation over it.

There exists an entire spectrum of optimisations;

- instruction level (peephole)
- expression level (Sethi-Ullman weights)
- local (basic blocks - instruction sequence with single entry and exit point, no jumps or labels)  
generally runs quickly, easy to validate
- global (looks at whole procedure)  
worse than linear complexity
- interprocedural (entire program)

## Loop Optimisations

- **loop-invariant code motion** (an instruction is loop-invariant if its operands can only arrive from outside the loop)
- detection of induction variables (variables that increase or decrease by a loop-invariant constant on each iteration)
- strength reduction (calculate induction variables by addition instead of multiplying other induction variables)
- control variable selection (replace an unused loop control variable with a induction variable that's actually used)

Note that in the example below, we can see some of these rules;

```

1  int P(int N, int M) {
2      int i, u, v, w, x, y;
3      int z = 0;
4      for (i = 0; i < N; i++) {
5          w = w + 10;
6          x = w * 10;
7          y = z * (w - x);
8          u = w + x + y + N + M;
9          v = v + u
10     }
11     return v;
12 }
```

- (1)  $y$  is constant (product of  $0 * ???$ ) (line 7)
- (2)  $w - x$  is dead code (multiplied by constant 0) (line 7)
- (3)  $y + N + M$  is loop invariant ( $y$  is constant,  $N, M$  come from outside) (line 8)
- (4)  $i, w, x$  are induction variables ( $w + x$  also is)
- (5)  $x$  increases by 100 every iteration (strength reduction) (line 6)
- (6)  $i$  is only used for loop control, another induction variable can be exit condition

## Intermediate Code

As previously mentioned, we should generate intermediate code on the traversal of the AST, instead of going directly to assembly. The art of optimising compiler design lies in designing an IR satisfying the following;

- represent all primitive instructions for execution
- easy to analyse and manipulate
- independent of target architecture

Appel argues that the initial IR should be a tree representation before performing instruction selection, and a control flow graph afterwards. The IR should assume infinite registers, and these will be mapped to physical registers after optimisation and colouring.

## Data Flow Analysis

The goal of data flow analysis is to deduce the program properties from the IR. It first analyses every instruction for its meaning, and then combines this information with the meaning of other instructions in order to derive a meaning for the program.

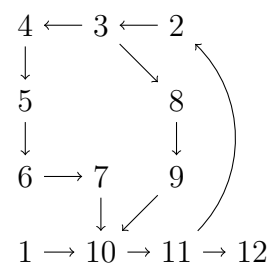
## Control Flow Graph

Consider the following source code, and its generated IR (with the node numbers on the left);

1	while (b < 10) {	1	[ 1] bra L2
2	if (b < a) {	2	L1:
3	a = a * 7;	3	[ 2] cmp b a
4	b = a + 1;	4	[ 3] bge L3
5	} else {	5	[ 4] mul #7 a
6	a = b - 1;	6	[ 5] mov a b
7	}	7	[ 6] add #1 b
8	}	8	[ 7] bra L4
		9	L3:
		10	[ 8] mov b a
		11	[ 9] sub #1 a
		12	L4:
		13	L2:
		14	[10] cmp b #10
		15	[11] blt L1
		16	[12]

This leads to the following table and control flow graph;

id	uses (reads)	defs (updates)	successors
1			10
2	a, b		3
3			4, 8
4	a	a	5
5	a	b	6
6	b	b	7
7			10
8	b	a	9
9	a	a	10
10	b		11
11			12, 2



We have the following terms in live variable analysis;

- a **point** is any location between adjacent nodes
- a **path** is a sequence of points, where  $p_{i+1}$  is the immediate successor of  $p_i$  in the CFG
- saying  $x$  is live at  $p$  is to say that the value of  $x$  could be **used** along some path starting at  $p$

For example, if we consider **b** after node 4, you will see that in node 5, the only successor to node 4, it is replaced with the value of **a**, and therefore "killed" - thus it will not reach any point after that.

This can be solved with a depth first search (for each variable, at every program point) - which is very expensive. It will need to be modified to detect cycles and stop, as well as detect when the value is updated, and halt the search there.

Let us define the following functions, where the difference between the two sets is **caused** by the instruction;

- $\text{LiveIn}(n)$  is the set of temporaries live immediately before node  $n$ 
  - it is live before  $n$  if it is live after node  $n$  (some later instruction reads it), unless  $n$  overwrites it
  - or it is used by node  $n$  (the instruction reads it)

$$\text{LiveIn}(n) = \text{uses}(n) \cup (\text{LiveOut}(n) - \text{defs}(n))$$

- $\text{LiveOut}(n)$  is the set of temporaries live immediately after node  $n$ 
  - a variable is live immediately after node  $n$  if it is live before any of  $n$ 's successors

$$\text{LiveOut}(n) = \bigcup_{s \in \text{succ}(n)} \text{LiveIn}(s)$$

However, writing this out for each node generates a system of simultaneous equations.

## 3rd February 2020

This lecture starts with almost half an hour on using *ANTLR4*. For anyone reading this, please use parser combinators for WACC.

## Semantic Analysis

Ideally, we want to check the legality of the program at compile time. However, there will be some checks that we cannot do, such as checking array bounds, and those checks will still need to be done at runtime.

The idea of semantic analysis is to take in an AST, and output errors (if any) or give back semantic attributes (such as type information) augmented to the AST, as well as a symbol table (identifier map). In many compilers, the goal is to do as much work as this stage as possible, allowing for much easier code generation.

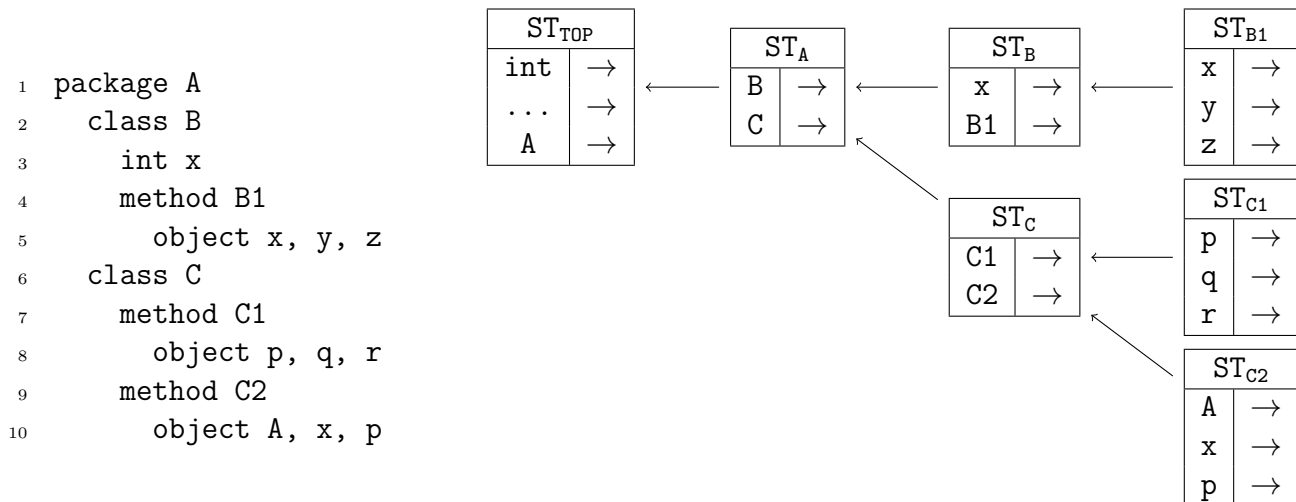
The rest of this lecture was him discussing one slide. Literally just going through what needs to be checked for six lines of code (and it's quite obvious).

- in an assignment, check if the identifier is declared as a variable
- check that the RHS of the assignment can be assigned to the LHS

5th February 2020

## Symbol Tables

Consider the symbol table as a dictionary or map (requires fast lookup) to hold data on identifiers. However it is represented, it should allow for scoping (thus allowing for redeclaration in an inner scope). One method for doing this is to have a global symbol table, with each entry holding auxiliary data on the scope. Another approach is to have a tree of symbol tables, with one for each lexical scope, pointing to the symbol table for the parent scope. The idea behind this is to first search in the local scope, then the parent scope, and so on, as it attempts to find the closest match for the identifier. A global scope is also added, holding predefined identifiers, such as functions, data types, and so on. When semantic analysis is done, there is also a global variable indicating the symbol table for the current lexical scope. For example, the code below has the following symbol table tree (note that the arrows ( $\rightarrow$  in each row represents additional information about that identifier - this can be predefined, in the case of predefined identifiers, or may refer to AST nodes if it is programmer defined);



In the example above, when we perform semantic analysis, we add **A** to the symbol table for the **TOP** scope, and then add auxiliary information into the entry, specifying that it is a package **and** points to the new symbol table for **A**'s scope. From here, when we see the class **B**, we add an entry into **A**'s symbol table, specifying that **B** is a class, and pointing to **B**'s symbol table. Note that the referencing to the parent symbol table is done as expected. In **B**, we see the integer declaration for **x**. This is added to the symbol table as a variable, and the type of the variable references the **int** type declared in the global scope.

In general, the symbol table is used by the AST nodes as follows;

- variable declaration

In the case of a variable declaration AST node, given the name of the type, and the name of the variable, we can check if type exists in **all accessible** scopes, check if the variable exists in the **local** scope. The error cases are as follows;

- identifier exists in the local scope (already declared)
- type not found (unknown type)
- type identifier found, but isn't a type
- type found but cannot be declared

Otherwise, add it to the current scope's symbol table, and give back this newly derived attribute to the AST.

- assignment

Note that in this case, we also want to recursively validate the RHS of the assignment (it also assumes that this augments the type of the RHS expression to the RHS' AST node). The error cases are as follows;

- the identifier is not found (hence it hasn't been declared or isn't accessible)
- the identifier is found, but isn't a variable
- the identifier is found, and is a variable, but the types aren't compatible

Otherwise, add the value in the symbol table to the AST node.

- function declaration

This starts off similar to the variable declaration check, but also must perform additional checks due to the scoping. After performing the declaration checks (almost the same, assuming no overloading is allowed), it then creates a new symbol table for the new scope, it also adds the new symbol table to the function object augmented to the AST (and added into the current symbol table). It then validates each of its parameters (into the new symbol table), and links the new objects (the "formals") into the function object.

- function call

On a function call, it must check if the function exists, and is a function (similar to how variable assignment checks all scopes). In terms of the parameters, it also checks that the number of parameters expected (formals), matches with the number of parameters supplied (actuals). Each of the supplied parameters are then validated, and then checked that the types match up; if everything is successful, then the function object is augmented onto the call AST node. This does **not** check the return type, that should be handled by the enclosing expression.

It's also important to note that, in practice, type checking should also allow for asymmetry (such as letting an integer be assigned to a double, but not the other way around). This can also include single inheritance (for example, if the LHS expected class A, but got class B, it should allow it if A is a superclass of B);

```
1 def compatible(TYPE lhs, TYPE rhs)
2     if lhs == rhs: return true
3     if rhs instanceof CLASS: return compatible(lhs, rhs.superclass)
4     ...
5     else: return false
```

## 7th February 2020

### Iteratively Solving Live Ranges

Continuing from the last lecture, we focus on the problem of computing the LiveIn and LiveOut sets. In a program with cyclic dependencies, i.e. with loops, this cannot be solved by evaluating the RHS of each set equation. A method of solving this would be to use an iterative algorithm is follows;

```
1 for n in CFG {
2     LiveIn(n) = { };
3     LiveOut(n) = { };
4 }
5 repeat {
6     for n in CFG {
7         LiveOut(n) = [union over s : succ(n)] LiveIn(s);
8         LiveIn(n) = uses(n) union (LiveOut(n) - defs(n));
9     }
10 } until LiveOut and LiveIn stop changing
```

We can claim that the algorithm must halt as the sets can only get larger (monotonicity), and therefore must stop at some point. If it does halt, which we claim it will, it must be because  $\text{LiveOut}(n)$  and  $\text{LiveIn}(n)$  don't change, which only happens when the LHS is equal to the RHS (hence satisfied). Additionally, since it halts as soon as it is satisfied, and only gets larger, we have the minimal set. Note that the complexity (the number of iterations) grows with the loop complexity of the code - however, this number is usually quite low.

## Interference Graph

From here, we have calculated where all the live ranges are, and can therefore use that to generate live ranges. For example, if we say  $\text{LiveOut}(p) = \{x, y, z\}$ , then we are saying the values of  $x, y, z$  can be used along some path starting from  $p$ . Using this information, we can construct an interference graph as follows;

- for each temporary  $t$ , check each node  $n$
- if  $t \in \text{LiveOut}(n)$ , then  $t$  interferes all temporaries in  $\text{LiveOut}(n)$

This can then be solved with some graph colouring algorithm, to map temporaries registers to physical registers.

## Loop Optimisations

The goal is to move loop invariant instructions out of the loop. For this part, we need to specify the following definitions;

- a definition  $d$  of a variable  $t$  is a statement which may assign to  $t$
- a definition  $d$  reaches a program point  $p$  if there exists a path from  $d$  to  $p$ , such that  $d$  is not killed along that path
- consider the following CFG nodes (which are all definitions of  $t$ );

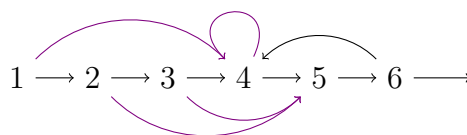
- $n : t := u_1 \oplus u_2$
- $n : t := u_1$
- $n : t := \text{CONSTANT}$

- a definition is loop-invariant if for every  $u \in \text{uses}(d)$ ; all the definitions of  $u$  that reach  $d$  are out of the loop OR only one definition of  $u$  reaches  $d$ , and it is also loop invariant.
- $\text{Gen}(n)$  is the set of definitions generated by the node  $n$   $\{n\}$
- $\text{Kill}(n)$  is the set of all definitions of  $t$  excluding  $n$
- $\text{ReachIn}(n)$  is the set of all definitions reaching the point before  $n$
- $\text{ReachOut}(n)$  is the set of all definitions reaching the point after  $n$

For example, looking at the following code (not really assembly, any jumps will be drawn in the graph - nodes are corresponding to line numbers);

```

1  x = 1
2  w = 100
3  z = 200
4  x = x + 1
5  y = w + z
6  if (x < 10)
```





Reaching definitions links a use of a variable back to where its value could've been generated. The data flow equations are as follows;

$$\text{ReachIn}(n) = \bigcup_{p \in \text{pred}(n)} \text{ReachOut}(p)$$

$$\text{ReachOut}(n) = \text{Gen}(n) \cup (\text{ReachIn}(n) - \text{Kill}(n))$$

The reach out set of a node is itself, combined with any definitions it has "inherited" from its predecessors (its ReachIn set) excluding anything it has killed. Similarly, this can be solved with the same iterative method used for the LiveIn and LiveOut sets (repeatedly applying the set equations until they converge). The same arguments apply for this iterative method. For our example above, the reaching definitions are as follows;

$$\text{ReachIn}(1) = \{\}$$

$$\text{ReachIn}(2) = \{1\}$$

$$\text{ReachIn}(3) = \{1, 2\}$$

$$\text{ReachIn}(4) = \{1, 2, 3, 4, 5\}$$

$$\text{ReachIn}(5) = \{2, 3, 4, 5\}$$

$$\text{ReachIn}(6) = \{2, 3, 4, 5\}$$

However, this gives us definitions we don't care about (for example, node 4 doesn't need node 2, 3, or 5, because it doesn't use any of those values). Therefore the relevant reaching definitions are as follows;

$$\text{ReachIn}(1) = \{\}$$

$$\text{ReachIn}(2) = \{\}$$

$$\text{ReachIn}(3) = \{\}$$

$$\text{ReachIn}(4) = \{1, 4\}$$

$$\text{ReachIn}(5) = \{2, 3\}$$

$$\text{ReachIn}(6) = \{4\}$$

all definitions (used) lie outside the loop

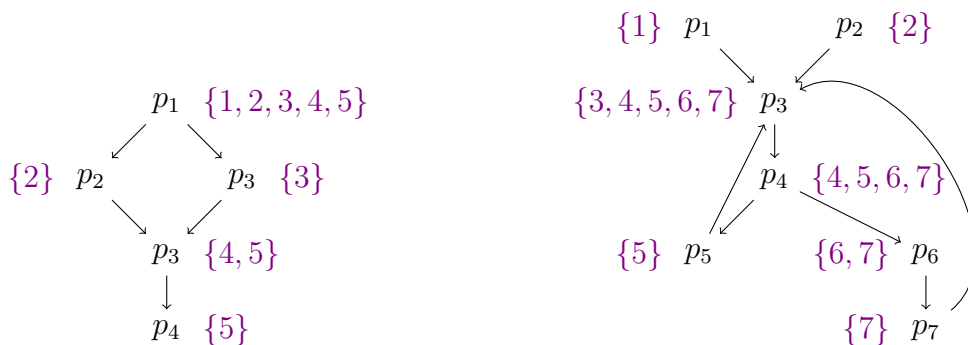
## Natural Loops and Dominators

A natural loop comes from sensible code, and it is a more subtle concept than just a loop in the control flow graph. Essentially, a natural loop in a CFG is a set of nodes  $S$ , including a header node  $h$ , that has the following properties, with the last point being the most important;

- there exists a path any node in  $S$  to  $h$
- there exists a path from  $h$  to every node in  $S$
- there is no path from a node outside  $S$  to a node in  $S$  that doesn't go through  $h$

this is obvious in sensible code; there is only one entry to a loop, you shouldn't be able enter a loop at some arbitrary point within the loop

We say that a node  $d$  dominates a node  $n$  if every path from the start node of the CFG to  $n$  must pass through  $d$  - a node dominates itself. In graphs below, we can see the set of nodes that each node dominates.



Note that in the first example,  $p_2$  and  $p_3$  don't dominate  $p_4$  (and thus  $p_5$ ), as it can go through either point.

We can define the dominance relation between two nodes based on predecessors. Given a start node  $s$ , a candidate dominator  $d$ , predecessor nodes  $p_1, \dots, p_k$  to  $n$ , we can figure out whether  $d$  dominates  $n$ .  $d$  dominates  $n$  if, and only if,  $d$  dominates all of the  $p_i$ . As any path from  $s$  (start) to  $n$  must go through one of the  $p_i$ , if  $d$  dominates all of them, all the paths from  $s$  to  $n$  **must** go through  $d$ . The argument can be made the other way around - if we know  $d$  dominates  $n$ , then we also know that  $d$  must dominate all  $p_i$ , otherwise there would be a path going through at least one  $p_i$  that doesn't go through  $d$ .

Let the set of nodes that dominate  $n$  be  $\text{Doms}(n)$ . We then have this system of simultaneous set equations;

$$\begin{aligned} \text{Doms}(s) &= \{s\} && \text{start node} \\ \text{Doms}(n) &= \{n\} \cup \bigcap_{p \in \text{preds}(n)} \text{Doms}(p) && \text{otherwise} \end{aligned}$$

The second equation is due to our goal of finding what dominators are common to all our predecessors. This is solved in the same way as before, however, instead of starting with an empty set, we start every set as the set of all nodes in the graph. Each iteration will make  $\text{Doms}(n)$  smaller, until at some point it converges, and the same arguments hold.

The idea of natural loops is linked to the dominance relation. A **back edge** is an edge from some node  $n$  to a node  $h$  that dominates  $n$ .

## 10th February 2020

### Type Representation

The representation typically falls into what the architecture provides.

- primitive types

Primitive types, such as booleans, integers, characters, typically fit into registers. Integers are typically represented in 2's complement, floating point numbers are typically represented in IEEE standard (see last year's **CO113**), and characters may be in any encoding (such as ASCII or Unicode).

- records / structs and arrays

Records or structs are fields grouped together (and may have different types or sizes), typically stored consecutively in memory. The fields are accessed via a constant byte offset from the start of the record. The fields do not have to be ordered in the same way they are declared - they can be rearranged by the compiler to be more optimal. On the other hand, arrays are of a fixed type, and bounds have to be checked dynamically (if the size can change) - the access is done at runtime by calculating offsets.

- objects

These are typically stored as references to memory allocated on the heap. The record on the heap will also consist of a reference to the class' method lookup table (containing the addresses for the method calls), as well as the data fields of the object. To access a field, we need to read the memory as follows;  $\text{Mem}[\text{Mem}[\text{@Reference}] + \text{Offset}]$ . When a call is made, the object reference is also implicitly given as a parameter.

While the scheme of using a method lookup table may seem complex, it allows for inheritance and overriding for free - it can put in the addresses of the superclass' methods into its own method lookup table. Dynamic binding is also possible, as we can point to the new location of object values from the original object reference.

## Program Address Space

On modern architectures, a compiled program is typically given a large virtual address space. This is normally divided into segments, and we consider the following three;

- code segment holds the compiled program code
- stack segment holds method calls and local variables  
typically when we make a call into a function we set the start of the frame with a reserved register (frame pointer), as the stack pointer can constantly change
- data segment holds global variables in the static area, and dynamic variables in the heap area  
global / static variables exist for the lifetime of a program's execution and are normally mapped to absolute positions in the static area of the data segment (can also include constants and method lookup tables)  
dynamic variables are normally mapped to locations in the heap area, and are allocated when the program calls some allocator function (`new` in Java, `malloc` in C, etc.), and de-allocated when explicitly done (`free` in C), or automatically via garbage collection.

**12th February 2020**

## Explicit Heap Allocation

When a block of memory of size  $N$  is requested, the allocation algorithm should do the following;

- search through a list of free memory blocks for a block of size  $N$  or greater
- if a block of exactly size  $N$  is found, remove the block from the list, and return the start address of the block
- if a block larger than  $N$  is found, split the block into  $N$  and a residual block, leave the residual block in the free list, and return the start address of the split block
- otherwise if there is no suitable block, request more memory from the OS (otherwise handle a failure)

When memory is deallocated, insert the freed block into the list of free blocks, and attempt to coalesce the returned block with the adjacent ones (to reduce fragmentation). However; manually doing this has many pitfalls, either from the programmer allocating memory and not freeing it after it is no longer needed (memory leaks), or freeing memory too early (still needed after freeing). An ideal system would automatically free the memory, but still allow the programmer to manually allocate memory.

## Garbage Collection

Many modern languages prevent programmers from directly manipulating memory. Garbage collection is one of the features used for this, as it automatically reclaims memory that is no longer used to be allocated later. A garbage collector must have correctness (it cannot collect live data), it must be fast, and it should also have low memory overhead. Some techniques are as follows;

- heap compaction (not GC)

While this isn't necessarily a garbage collection technique, it is done after garbage collection - as that process may fragment the heap. This aims to move all the live blocks together, thus allowing for one large block of free memory, in three steps; it first marks the live blocks, then co-locates it together, and finally updates the pointers to the live blocks.

- reference counting

This counts the uses of something on the heap. For example, in the assignment  $p = q$ , if  $q$  is on the heap, the reference count of  $q$  should be incremented, and if  $p$  is on the heap, the reference count of  $p$  should be decremented - if it is 0, then it should be freed. However, a naive implementation of this cannot deal with cyclic references.

- mark-sweep

This performs garbage collection in two phases - it first marks all the live blocks, in one pass, and then scans all the blocks, freeing any dead blocks and unmarking any live blocks. This requires one bit to indicate a block's status in each block.

```

1 def new(size):
2     if (freeList empty):
3         mark_sweep()
4         allocate(size)
5 def mark_sweep():
6     foreach heapRef: # all blocks reachable from non-heap references
7         mark(heapRef.block)
8     sweep()
9     if (freeList empty):
10        error
11 def mark(block):
12     if (block.marked == 0): # not yet marked
13         block.marked = 1
14         foreach childRef: # iterates through children
15             mark(childRef.block)
16 def sweep():
17     block = heapStart
18     while block < heapEnd:
19         if (block.marked == 0): # dead block
20             deallocate(block)
21         else
22             block.marked = 0
23         block = block + block.size

```

However, this is a recursive process and can be costly in terms of memory. Pointer reversal (*Schorr and Waite*) is a method to do this without additional stack space. When visiting a child block  $C$  of parent  $P$ , the parent of  $P$ 's pointer is stored in  $P$ , in place of where the pointer to  $C$  was stored. After the visit to  $C$  is complete, it restores the pointer in  $P$ , and can be swapped with the location of the next pointer in  $C$ .

- two-space

The heap is split into two spaces (From and To), we only allocate from the From space, and when the From space is full, we copy all the live blocks from the From space to the To space - the roles are then swapped. This is fast and automatically compacts the heap, but it wastes half the memory.

- generational

This works on the idea that newly created objects tend to die quickly, and older objects often survive longer (as they have already survived for a while). The heap is divided into generations, and allocates blocks from the youngest generation (which is also collected more frequently), and move blocks to an older generation based on age.

## Debuggers and Profilers

In a post-mortem dump for a program, we often want the information to be as close to the programming language as possible, to allow for the user to fix an error. The Stack Traceback should return the name of the method, and the call stack (as well as local variables if possible). However, since local variables may be on the stack or in registers, it's important for the debugger to know where those values are located.

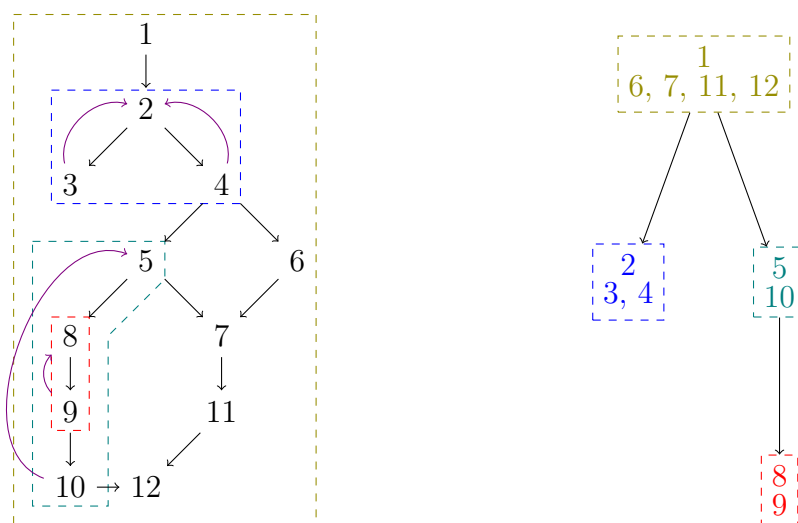
In terms of interactive debuggers, we need support from the architecture / OS to set breakpoints on the machine instructions. This can be more difficult with optimising compilers.

Profilers allow the programmer to identify where most of the time is being spent. A method is to look at the program counter periodically, and relate that to the program method that is associated with it (and increment its count).

## 14th February 2020

### Natural Loops

Given a back edge  $(n, h)$ , such that  $h$  dominates  $n$ , we can define the natural loop set  $S$  (with header  $h$ ) as the set of nodes  $x$  such that  $x$  is dominated by  $h$ , and there is a path from  $x$  to  $n$ , which doesn't contain  $h$ . It's also important to note that there can exist some ambiguity, for example in the second CFG (from the last lecture) there are two back edges, and it can correspond to the same loop, or two loops with the same entry point. In the example below, we demonstrate a nested loop (the back edges are highlighted in violet);



Suppose we have loops  $A$  and  $B$ , with headers  $a$  and  $b$  ( $a \neq b$ ), and  $b$  is in  $A$  ( $b \in A$ ). Then all the nodes of  $B$  must be a proper subset of the nodes of  $A$  ( $B \subset A$ ).  $B$  is therefore nested within  $A$  ( $B$  is the inner loop). By identifying these loops (and generating a control tree), we can reconstruct the structured control flow from the CFG. From here, we can lift loop-invariant code to a point just before the header. However, if a header has multiple predecessors, then we insert a pre-header, and move the code there. Note that we can't always lift loop-invariant code (the line  $t = a \ 0 \ b$  would be moved to the dashed line before the loop header);

(A) it is fine to move code out

(B) loop invariant node does not dominate all loop exits

you can reach the exit without passing through the loop invariant code

(C) more than one definition of  $t$  in the loop

$t$  can be redefined in the loop

(D)  $t$  is LiveOut from the loop's header

there is another definition of  $t$  that is produced before the header and used on the way in - by moving it out, we kill that value

```

1 L0: # EXAMPLE A
2   t = 0
3   -----
4 L1:
5   i = i + 1
6   t = a 0 b
7   M[i] = t
8   if i<N goto L1
9 L2:
10  x = t

```

In this example, it is fine to move the instruction out of the loop.

```

1 L0: # EXAMPLE B
2   t = 0
3   -----
4 L1:
5   if i<N goto L2
6   i = i + 1
7   t = a 0 b
8   M[i] = t
9   goto L1
10 L2:
11  x = t

```

This would give  $x$  the wrong value if it fails  $i<N$  on the first iteration.

```

1 L0: # EXAMPLE C
2   t = 0
3   -----
4 L1:
5   i = i + 1
6   t = a 0 b
7   M[i] = t
8   t = 0
9   M[j] = t
10  if i<N goto L1
11 L2:

```

This would leave  $M[i]$  with the wrong value on the second iteration.

```

1 L0: # EXAMPLE D
2   t = 0
3   -----
4 L1:
5   M[j] = t
6   i = i + 1
7   t = a 0 b
8   M[i] = t
9   t = 0
10  if i<N goto L1
11 L2:

```

This would leave  $M[j]$  with the wrong value on the first iteration.

Therefore, the conditions for moving a CFG node  $d : t = a \oplus b$  out of a loop are as follows;

1. loop invariant - all reaching definitions used by  $d$  must occur outside the loop

this can be done with reaching definitions data flow analysis

2. loop invariant node must dominate all loop exits

use dominators analysis

3. there must be just one definition of  $t$  in the loop

trivial to count

4.  $t$  must not be LiveOut from the loop's pre-header

this can be done with live variables data flow analysis

## Static Single Assignment (unassessed)

This is a transformation the compiler applies to the code to ensure no variable is ever redefined (and therefore defined exactly once). Note that  $p(t_1, t_2)$  represents the  $\phi$  function  $\phi(t_1, t_2)$ . Looking at example B from before, we can put this into SSA as follows (notice the additional instruction for  $t_3$ );

```

1 L0: # phi inserted
2   t1 = 0
3   -----
4 L1:
5   t3 = p(t1, t2)
6   if i<N goto L2
7   i = i + 1
8   t2 = a 0 b
9   M[i] = t2
10  goto L1
11 L2:
12  x = t3

```

```

1 L0: # code hoisted
2   t1 = 0
3   t2 = a 0 b
4 L1:
5   t3 = p(t1, t2)
6   if i<N goto L2
7   i = i + 1
8   M[i] = t2
9   goto L1
10 L2:
11  x = t3

```

```

1 L0: # phi propagated
2   t1 = 0
3   t2 = a 0 b
4   t3 = t1
5 L1:
6   if i<N goto L2
7   i = i + 1
8   M[i] = t2
9   t3 = t2
10  goto L1
11 L2:
12  x = t3

```

The third stage is reached when generating code from SSA. In examples C and D, it's much simpler than in B which is due to a conditional - a simple renaming is sufficient.

**17th February 2020**

**Tutorial**

## Lexical Analysis

Note that when we are given an alphabet, we only consider the characters in that alphabet (whereas normally we'd consider the entire character set in actual use). Another tip, when dealing with regex meta-characters, is to cast those into standard characters to make it easier to work with, and replace them at the end (escaping if needed).

10. For the alphabet  $\{A, B\}$  give a regular expression for all strings containing an even number of As and an even number of Bs.

$(AA \mid BB \mid (AB|BA) (AA|BB)^* (AB|BA))^*$

The interesting case is the third (outer) alternative - if there is a string that uses each character once, there must be another string of two at some point that uses each character once.

15. This was a trick question - because we cannot count the number of 0s, and the number of 1s (since regular expressions cannot count).
16. The idea here was to see what the cycles in the DFA do (and what is in the accepting state).
17. In this question, because it was finite (due to no cycles) - it was possible to enumerate the values it could generate (since it was binary, it made sense to check the values in order).

## LR Parsing

In the case of context free grammars, we are able to utilise recursion, which mirrors how programming language structure is defined.

1. This continues from the limitation of regular expressions (question 5 of the last tutorial). We want to obtain the CFG for the set of strings where there are  $n$  0s followed by  $n$  1s.

We can start by looking at what we want to capture; 01, 0011, 000111, and so on. You'll notice that (other than the first case,  $n = 1$ ), the cases build on the case before it, due to the recursive nature, by adding a 0 to the start, and a 1 afterwards.

$S \rightarrow 0 \ 1 \mid 0 \ S \ 1$

3. In this question, we typically assume that the start symbol is the first rule listed. From that rule, we look at the alternatives to it, and attempt them to find one that fits (looking at the terminals (if possible) is a good way to "guess" the alternative we need).
4. Show all binary strings generated in the following grammar are divisible by 3;

$\text{num} \rightarrow 11 \mid 1001 \mid \text{num } 0 \mid \text{num num}$

This is done via numerical values for nodes in the parse tree - going through each alternative;

rule 1: 11 is decimal 3, and is trivially divisible by 3

rule 2: 1001 is decimal 9, and is trivially divisible by 3

rule 3:  $\text{num } 0$  (from here, we use induction and assume  $\text{num}$  is divisible by 3) - this multiplies  $\text{num}$  by 2, and if  $\text{num}$  is a multiple of 3, the new number also remains as a multiple of 3

rule 4:  $\text{num num}$  (we assume both  $\text{nums}$  are multiples of 3) - let the length of the second  $\text{num}$  be  $k$  bits, and therefore this multiplies the first  $\text{num}$  by  $2^k$ , and adds the second  $\text{num}$  - multiplying keeps it as a multiple of 3, and adding a multiple of 3 to a multiple of 3 is a multiple of 3

12. Note that we check whether the grammar is LR(1) by looking at the constructed parsing table - if it has any conflicts, then it cannot be LR(1).

$$C \rightarrow C \ i \ o \mid i \ o$$

Recall that we need to add an auxiliary state,  $C' \rightarrow C \ \$$ . When we have a non-terminal in a state, being the "next" symbol to read, we can compute the  $\epsilon$ -transitions into the same state (as it would get collapsed anyways).

state 0:

$$\begin{array}{ll} C' \rightarrow \bullet C, \$ & \\ C \rightarrow \bullet C \ i \ o, \$ & \epsilon \text{ from first item} \\ C \rightarrow \bullet C \ i \ o, i & \epsilon \text{ from item above} \\ C \rightarrow \bullet i \ o, i & \epsilon \text{ from item above} \\ C \rightarrow \bullet i \ o, \$ & \epsilon \text{ from first item} \end{array}$$

state 1:

$$\begin{array}{l} C' \rightarrow C \bullet, \$ \\ C \rightarrow C \bullet \ i \ o, \$ \\ C \rightarrow C \bullet \ i \ o, i \end{array}$$

state 2:

$$\begin{array}{l} C \rightarrow C \ i \bullet \ o, \$ \\ C \rightarrow C \ i \bullet \ o, i \end{array}$$

state 3:

$$\begin{array}{l} C \rightarrow C \ i \ o \bullet, \$ \\ C \rightarrow C \ i \ o \bullet, i \end{array}$$

state 4:

$$\begin{array}{l} C \rightarrow i \bullet \ o, i \\ C \rightarrow i \bullet \ o, \$ \end{array}$$

state 5:

$$\begin{array}{l} C \rightarrow i \ o \bullet, i \\ C \rightarrow i \ o \bullet, \$ \end{array}$$

This gives the following DFA

