CMSC 430 – Today …

- Goals for course
- What is a compiler and briefly how does it work?
  - Review everything you “should” know from 330 and before
    - This looks like a lot of material (and it is), but most of this should be review. (Future lectures will slow down quite a bit!)
    - BNF and context free languages
    - FSA and regular grammars

Compilers

- What is a **compiler**?
  - A program that translates an *executable* program in one language into an *executable* program in another language
  - The compiler should improve the program, *in some way*

- What is an **interpreter**?
  - A program that reads an *executable* program and produces the results of executing that program

- C is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
  - which are then interpreted (Java VM is not a piece of hardware)

This course deals mainly with **compilers**
Many of the same issues arise with **interpreters**
Two Goals for a compiler

- The compiler must preserve the meaning of the program being compiled.
  - Is this true?
  - What does the term “meaning” apply to in this statement?
- The compiler must improve the source code in a discernable way.
  - What are some ways to improve code without improving performance?
- What other non-standard compilers can you think of?
  - HTML
  - Postscript
  - Excel macros

Why study compilation?

- Compilers are important system software components
  - They are intimately interconnected with architecture, systems, programming methodology, and language design
- Compilers include many applications of theory to practice
  - Scanning, parsing, static analysis, instruction selection
- Many practical applications have embedded languages
  - Web pages, commands, macros, formatting tags …
- Many applications have generalized input formats that look like languages, to increase their generality
  - Matlab, Mathematica,
- Writing a compiler exercises practical algorithmic & engineering issues
  - Approximating hard problems
  - Emphasis on efficiency & scalability
Intrinsic merit

- Compiler construction poses challenging and interesting problems:
  - Compilers must do a lot but also run fast
  - Compilers have primary responsibility for run-time performance
  - Compilers are responsible for making it acceptable to use the full power of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
  - Compilers must hide that complexity from the programmer
  - Success requires mastery of complex interactions

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus
Program structure

Syntax
- What a program looks like
- BNF (context free grammars) - a useful notation for describing syntax.

Semantics
- Execution behavior
  - Static semantics - Semantics determined at compile time:
    - var A: integer; Type and storage for A
    - int B[10]; Type and storage for array B
    - float MyProcC(float x;float y){...}; Function attributes
  - Dynamic semantics - Semantics determined during execution:
    - X = "ABC" SNOBOL4 example: X a string
    - X = 1 + 2; X an integer
    - :(X) X an address; Go to label X

Translation environments
Implications

• Must recognize legal (and illegal) programs
• Must generate correct code
• Must manage storage of all variables (and code)
• Must agree with OS & linker on format for object code

*Big step up from assembly language—use higher level notations*

---

**Traditional Two-pass Compiler**

Implications

• Use an intermediate representation (IR)
• Front end maps legal source code into IR
• Back end maps IR into target machine code
• Admits multiple front ends & multiple passes *(better code)*

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC
**A Common Fallacy**

Can we build \( n \times m \) compilers with \( n+m \) components?
- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

*Limited success in systems with very low-level IRs*

---

**The Front End**

```
Source code
  \[\rightarrow\]
  Scanner
  \[\rightarrow tokens\]
  Parser
  \[\rightarrow Errors, IR\]
```

**Responsibilities**
- Recognize legal (and illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the back end
- Much of front end construction can be automated
  - The big success story of compiler design
The Front End

Scanner

- Maps character stream into words—the basic unit of syntax
- Produces words & their parts of speech
  \[ x = x + y; \] becomes
  \[ <id,x> <oper,=> <id,x> <oper,+) <id,y> <endtoken,;> \]
  
  > word \( \equiv \) lexeme, part of speech \( \equiv \) tokentype
  > In casual speech, we call the pair a token
- Typical tokens include number, identifier, +, -, while, if
- Scanner eliminates white space (including comments)
- Speed is important

Parser

- Recognizes context-free syntax & reports errors
- Guides context-sensitive (“semantic”) analysis (type checking)
- Builds IR for source program

\[ \text{Hand-coded parsers are fairly easy to build} \]
\[ \text{Most books advocate using automatic parser generators} \]
The Front End

Context-free syntax is specified with a grammar

\[ \text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \]
\[ \text{SheepNoise} \rightarrow \ baa \]

This grammar defines the set of noises that a sheep makes under normal circumstances.

It is written in a variant of Backus–Naur Form (BNF)

Formally, a grammar \( G = (S,N,T,P) \)

- \( S \) is the start symbol
- \( N \) is a set of non-terminal symbols
- \( T \) is a set of terminal symbols or words
- \( P \) is a set of productions or rewrite rules \( \left( P : N \rightarrow N \cup T \right) \)

Example due to Dr. Scott K. Warren

---

The Front End

Context-free syntax can be put to better use

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr} \ \text{op} \ \text{term} \)
3. \( \text{term} \rightarrow \text{number} \)
4. \( \text{op} \rightarrow + \)
5. \( \text{op} \rightarrow - \)

\( S = \text{goal} \)
\( T = \{ \text{number}, \text{id}, +, - \} \)
\( N = \{ \text{goal}, \text{expr}, \text{term}, \text{op} \} \)
\( P = \{ 1, 2, 3, 4, 5, 6, 7 \} \)

- This grammar defines simple expressions with addition & subtraction over “number” and “id”
- This grammar, like many, falls in a class called “context-free grammars”, abbreviated CFG
The Front End

Given a CFG, we can derive sentences by repeated substitution:

<table>
<thead>
<tr>
<th>Production</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td></td>
</tr>
<tr>
<td>1 expr</td>
<td></td>
</tr>
<tr>
<td>2 expr op term</td>
<td></td>
</tr>
<tr>
<td>5 expr op y</td>
<td></td>
</tr>
<tr>
<td>7 expr - y</td>
<td></td>
</tr>
<tr>
<td>2 expr op term - y</td>
<td></td>
</tr>
<tr>
<td>4 expr op 2 - y</td>
<td></td>
</tr>
<tr>
<td>6 expr + 2 - y</td>
<td></td>
</tr>
<tr>
<td>3 term + 2 - y</td>
<td></td>
</tr>
<tr>
<td>5 x + 2 - y</td>
<td></td>
</tr>
</tbody>
</table>

To recognize a valid sentence in some CFG, we reverse this process and build up a parse.

The Front End

A parse can be represented by a tree (parse tree or syntax tree):

```
       goal
       /   |
      expr  \
     /   |
    expr op term
    /   \
  expr op term
     /   \
 term op term
   /   \
<id,x> term
```

This contains a lot of unneeded information.

1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | -
The Front End

Compilers often use an abstract syntax tree

![Diagram of abstract syntax tree](image)

The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

ASTs are one form of intermediate representation (IR)

The Back End

Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been much less successful in the back end
The Back End

Instruction Selection
- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  > *ad hoc* methods, pattern matching, dynamic programming

This was the problem of the future in 1978
  > Spurred by transition from PDP-11 to VAX-11
  > Orthogonality of RISC simplified this problem

Register allocation
- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete (1 or \(k\) registers)

Compilers approximate solutions to NP-Complete problems
The Back End

Instruction Scheduling
- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases
Heuristic techniques are well developed

Traditional Three-pass Compiler

Code Improvement (or Optimization)
- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
- Note that datatype input to optimizer is same as output
  - Optimization is often an optional phase
The Optimizer (or Middle End)

Modern optimizers are structured as a series of passes

Typical Transformations
- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Example
- Optimization of Subscript Expressions in Fortran

[Do you remember this from CMSC 330?]

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize a multiplication is generated here?

compute addr(A(0,J)
DO I = 1, M
A(I,J) = A(I,J) + C
ENDDO

DO I = 1, M
add 1 to get addr(A(I,J)
A(I,J) = A(I,J) + C
ENDDO
**Modern Restructuring Compiler**

- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining

Will only briefly discuss this. Subject to later courses.

---

**Role of the Run-time System**

- Memory management services
  - Allocate
    - Heap or stack frame
  - Deallocate
  - Collect garbage
- Run-time type checking
- Error processing
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization
Compiler jargon we will be using -- Major stages

- **Lexical analysis (Scanner)**: Breaking a program into primitive components, called tokens (identifiers, numbers, keywords, ...) We will see that regular grammars and finite state automata are formal models of this.

- **Syntactic analysis (Parsing)**: Creating a syntax tree of the program. We will see that context free grammars and pushdown automata are formal models of this.

- **Symbol table**: Storing information about declared objects (identifiers, procedure names, ...)

- **Semantic analysis**: Understanding the relationship among the tokens in the program.

- **Optimization**: Rewriting the syntax tree to create a more efficient program.

- **Code generation**: Converting the parsed program into an executable form.

- We will briefly look at scanning and parsing. A full treatment of compiling is beyond scope of this course.

---

Review of grammars from CMSC 330: BNF grammars

**Nonterminal**: A finite set of symbols: `<sentence> <subject> <predicate> <verb> <article> <noun>`

**Terminal**: A finite set of symbols: the, boy, girl, ran, ate, cake

**Start symbol**: One of the nonterminals: `<sentence>`

**Rules (productions)**: A finite set of replacement rules:

- `<sentence>` ::= `<subject> <predicate>`
- `<subject>` ::= `<article> <noun>`
- `<predicate>` ::= `<verb> <article> <noun>`
- `<verb>` ::= ran | ate
- `<article>` ::= the
- `<noun>` ::= boy | girl | cake

**Replacement Operator**: Replace any nonterminal by a right hand side value using any rule (written ⇒)
Example BNF sentences

\[ \text{<sentence>} \Rightarrow \text{<subject>} \text{<predicate>} \quad \text{First rule} \]
\[ \Rightarrow \text{<article>} \text{<noun>} \text{<predicate>} \quad \text{Second rule} \]
\[ \Rightarrow \text{the <noun>} \text{<predicate>} \quad \text{Fifth rule} \]
\[ \ldots \Rightarrow \text{the boy ate the cake} \]

Also from \text{<sentence>} you can derive
\[ \Rightarrow \text{the cake ate the boy} \]

Syntax does not imply correct semantics

Note:

Rule \text{<A>} ::= \text{<B><C>}

This BNF rule also written with equivalent syntax:
\[ \text{A} \rightarrow \text{BC} \]

Languages

Any string derived from the start symbol is a sentential form.

- **Sentence**: String of terminals derived from start symbol by repeated application of replacement operator
- A language generated by grammar \( G \) (written \( L(G) \)) is the set of all strings over the terminal alphabet (i.e., sentences) derived from start symbol.
- That is, a language is the set of sentential forms containing only terminal symbols.
**Derivations**

- A derivation is a sequence of sentential forms starting from start symbol.
- Derivation trees:
  - Grammar: \( B \rightarrow 0B \mid 1B \mid 0 \mid 1 \)
  - Derivation: \( B \Rightarrow 0B \Rightarrow 01B \Rightarrow 010 \)
- From derivation get parse tree
- But derivations may not be unique
  - \( S \rightarrow SS \mid (S) \mid (\) \)
  - \( S \Rightarrow SS \Rightarrow (SS) \Rightarrow (SS)(S) \Rightarrow (SS)(\) \)
  - Different derivations but get the same parse tree

**Ambiguity**

- But from some grammars you can get 2 different parse trees for the same string: \( ()()() \)
- Each corresponds to a unique derivation:
  - \( S \Rightarrow SS \Rightarrow SSS \Rightarrow SS()S \Rightarrow SS()() \)
- A grammar is ambiguous if some sentence has 2 distinct parse trees.
- We desire unambiguous grammars to understand semantics.
Role of \( \lambda \) (or \( \epsilon \))

How to characterize strings of length 0? – Semantically it makes sense to consider such strings.

1. In BNF, \( \epsilon \)-productions: \( S \rightarrow SS \mid (S) \mid () \mid \epsilon \)

Can always delete them in grammar. For example:

\[
\begin{align*}
  X & \rightarrow abYc \\
  Y & \rightarrow \epsilon
\end{align*}
\]

Delete \( \epsilon \)-production and add production without \( \epsilon \):

\[
\begin{align*}
  X & \rightarrow abYc \\
  X & \rightarrow abc
\end{align*}
\]

2. In FSA - \( \lambda \) moves means that in initial state, without input you can move to final state.

Syntax can be used to determine some semantics

During Algol era, thought that BNF could be used for semantics of a program:

What is the value of: \( 2 * 3 + 4 * 5 \)?

(a) 26
(b) 70
(c) 50

All are reasonable answers? Why?
Usual grammar for expressions

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T * P | P \]
\[ P \rightarrow i | (E) \]

“Natural” value of expression is 26
Multiply 2 \times 3 = 6
Multiply 4 \times 5 = 20
Add 6 + 20 = 26

But the “precedence” of operations is only a convention

Grammar for 70
\[ E \rightarrow E * T | T \]
\[ T \rightarrow T + P | P \]
\[ P \rightarrow i | (E) \]

Grammar for 50
\[ E \rightarrow E + T | E * T | T \]
\[ T \rightarrow i | (E) \]

All 3 grammars generate exactly the same language, but each has a different semantics (i.e., expression value) for most expressions. All are unambiguous.

Draw parse tree of expression 2\times3+4\times5 for each grammar
**Derivations**

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- **Leftmost derivation** — replace leftmost NT at each step
- **Rightmost derivation** — replace rightmost NT at each step

These are the two systematic derivations

*(We don’t care about randomly-ordered derivations!)*

The example on the preceding slide was a leftmost derivation

- Of course, there is a rightmost derivation
- Interestingly, it turns out to be different

---

**A More Useful Grammar**

To explore the uses of CFGs, we need a more complex grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Expr</td>
</tr>
<tr>
<td>Expr</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>Op</td>
<td>+</td>
</tr>
<tr>
<td>Op</td>
<td>−</td>
</tr>
<tr>
<td>Op</td>
<td>*</td>
</tr>
<tr>
<td>Op</td>
<td>/</td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>&lt;id,x&gt;</td>
<td>Op Expr</td>
</tr>
<tr>
<td>&lt;id,x&gt;</td>
<td>Expr</td>
</tr>
<tr>
<td>&lt;id,x&gt;</td>
<td>&lt;num,2&gt; Op Expr</td>
</tr>
<tr>
<td>&lt;id,x&gt;</td>
<td>&lt;num,2&gt; * Expr</td>
</tr>
<tr>
<td>&lt;id,x&gt;</td>
<td>&lt;num,2&gt; / Expr</td>
</tr>
</tbody>
</table>

We denote this: \( Expr \Rightarrow^* \text{id} \text{−} \text{num} \text{−} \text{id} \)

- Such a sequence of rewrites is called a derivation
- Process of discovering a derivation is called parsing
### The Two Derivations for \( x - 2 * y \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; Op Expr</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; - Expr</td>
</tr>
<tr>
<td>2</td>
<td>&lt;id,x&gt; - Expr Op Expr</td>
</tr>
<tr>
<td>3</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; Op Expr</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Expr</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

**Leftmost derivation**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>4</td>
<td>Expr Op &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>3</td>
<td>Expr Op &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>6</td>
<td>Expr - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

**Rightmost derivation**

In both cases, \( Expr \Rightarrow^* id - num * id \)

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

---

### Derivations and Parse Trees

#### Leftmost derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; Op Expr</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; - Expr</td>
</tr>
<tr>
<td>2</td>
<td>&lt;id,x&gt; - Expr Op Expr</td>
</tr>
<tr>
<td>3</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; Op Expr</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Expr</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

This evaluates as \( x - (2 * y) \)
Derivations and Parse Trees

Rightmost derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>4</td>
<td>Expr Op &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>2</td>
<td>Expr Op Expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>3</td>
<td>Expr Op &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>6</td>
<td>Expr – &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

This evaluates as \((x - 2) \cdot y\)

Derivations and Precedence

These two derivations point out a problem with the grammar:

It has no notion of precedence, or implied order of evaluation

To add precedence

- Create a non-terminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first
- Subtraction and addition, next
**Derivations and Precedence**

Adding the standard algebraic precedence produces:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal ( \rightarrow ) Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr ( \rightarrow ) Expr + Term</td>
</tr>
<tr>
<td>3</td>
<td>Expr ( \rightarrow ) Expr – Term</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
</tr>
<tr>
<td>5</td>
<td>Term ( \rightarrow ) Term * Factor</td>
</tr>
<tr>
<td>6</td>
<td>Term ( \rightarrow ) Term / Factor</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
</tr>
<tr>
<td>8</td>
<td>Factor ( \rightarrow ) number</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>id</td>
</tr>
</tbody>
</table>

This grammar is slightly larger:
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses our example:

The rightmost derivation

This produces \( x - (2 * y) \), along with an appropriate parse tree. Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.
Ambiguous Grammars

Our original expression grammar had other problems

This grammar allows multiple leftmost derivations for $x - 2 \times y$
- Hard to automate derivation if > 1 choice
- The grammar is ambiguous

Classic example — the if-then-else problem

This ambiguity is entirely grammatical in nature
**Ambiguity**

This sentential form has two derivations:

\[
\text{if } \text{Expr}_1 \text{ then if } \text{Expr}_2 \text{ then } \text{Stmt}_1 \text{ else } \text{Stmt}_2
\]

- Production 1:
  - **E1**
  - **S1**
  - **E2**
  - **S2**

- Production 2:
  - **E1**
  - **S1**
  - **E2**
  - **S2**

**Removing the ambiguity**

- Must rewrite the grammar to avoid generating the problem.
- Match each `else` to innermost unmatched `if` (common sense rule).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>Stmt</code> → <code>WithElse</code></td>
</tr>
<tr>
<td>2</td>
<td>`</td>
</tr>
<tr>
<td>3</td>
<td><code>WithElse</code> → <code>if Expr then WithElse else WithElse</code></td>
</tr>
<tr>
<td>4</td>
<td>`</td>
</tr>
<tr>
<td>5</td>
<td><code>NoElse</code> → <code>if Expr then Stmt</code></td>
</tr>
<tr>
<td>6</td>
<td>`</td>
</tr>
</tbody>
</table>

**Intuition:** A `NoElse` always has no else on its last cascaded `else if` statement.

With this grammar, the example has only one derivation.

CMSC430 Spring 2007
**Ambiguity (Motskau’s Grammar)**

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each `else` to innermost unmatched `if` *(common sense rule)*

With this grammar, the example has only one derivation

```
Stmt    \to WithElse
| NoElse
WithElse \to if Expr then WithElse else Stmt
| OtherStmt
NoElse  \to if Expr then Stmt
```

*Intuition: a statement with an else always has a WithElse in the then part*

**Deeper Ambiguity**

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

```
a = f(17)
```

In many Algol-like languages, `f` could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar
Ambiguity - the Final Word

Ambiguity arises from two distinct sources
- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity
- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  > Knowledge of declarations, types, ...
  > Accept a superset of $L(G)$ & check it by other means
  > This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar
- Parsing techniques that “do the right thing”
  → i.e., always select the same derivation

---

Classes of grammars

**BNF:** Backus-Naur Form - Context free - Type 2 - Already described

**Regular grammars:** subclass of BNF - Type 3:
- BNF rules are restricted: $A \rightarrow t N \mid t$
- where: $N$ = nonterminal, $t$ = terminal

**Examples:**
- Binary numbers: $B \rightarrow 0 B \mid 1 B \mid 0 \mid 1$
- Identifiers:
  - $I \rightarrow a L \mid b L \mid c L \ldots \mid z L \mid a \ldots y \mid z$
  - $L \rightarrow 1 L \mid 2 L \ldots \mid 9 L \mid 0 L \mid 1 \ldots \mid 9 \mid 0 \mid a L \mid b L \mid c L$
  - $\ldots \mid z L \mid a \ldots y \mid z$

---

Example string: $ab7d$
Other classes of grammars

The context free and regular grammars are important for programming language design. We study these in detail.

Other classes have theoretical importance, but not in this course:

- **Context sensitive grammar**: Type 1 - Rules: \( \alpha \rightarrow \beta \) where \( |\alpha| \leq |\beta| \)
  [That is, length of \( \alpha \) is less than or equal to length of \( \beta \), i.e., all sentential forms are length non-decreasing]

- **Unrestricted, recursively enumerable**: Type 0 - Rules: \( \alpha \rightarrow \beta \). No restrictions on \( \alpha \) and \( \beta \).

Finite state automaton

A finite state automaton (FSA) is a graph with directed labeled arcs, two types of nodes (final and non-final state), and a unique start state:

This is also called a state machine.

What strings, starting in state A, end up at state C?

The language accepted by machine M is set of strings that move from start node to a final node, or more formally: \( T(M) = \{ \omega | \delta(A,\omega) = C \} \) where A is start node and C a final node.
More on FSAs
An FSA can have more than one final state:

Deterministic FSAs

Deterministic FSA: For each state and for each member of the alphabet, there is exactly one transition.

Non-deterministic FSA (NDFSA): Remove restriction.

At each node there is 0, 1, or more than one transition for each alphabet symbol.

A string is accepted if there is some path from the start state to some final state.

Example nondeterministic FSA (NDFSA):
01 is accepted via path: ABD
even though 01 also can take the paths: ACC or ABC
and C is not a final state.
Non-deterministic Finite Automata

Each RE corresponds to a deterministic finite automaton (DFA)
- May be hard to directly construct the right DFA

What about an RE such as \((a | b)^* \text{abb}\)?

This is a little different
- \(S_3\) has a transition on \(\varepsilon\)
- \(S_1\) has two transitions on \(a\)

This is a non-deterministic finite automaton (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string \(x\) iff \(\exists\) a path though the transition graph from \(s_0\) to a final state such that the edge labels spell \(x\)
- Transitions on \(\varepsilon\) consume no input
- To “run” the NFA, start in \(s_0\) and guess the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts \(x\) then accept

Why study NFAs?
- They are the key to automating the RE \(\rightarrow\) DFA construction
- We can paste together NFAs with \(\varepsilon\)-transitions
**Relationship between NFAs and DFAs**

DFA is a special case of an NFA

- DFA has no \( \epsilon \) transitions
- DFA’s transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

> Obviously

NFA can be simulated with a DFA (less obvious)

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

---

**Equivalence of FSA and NDFSA**

Important early result:

NDFSA = DFSA

Let subsets of states be states in DFSA.

Keep track of which subset you can be in.

Any string from \( \{A\} \) to either \( \{D\} \) or \( \{CD\} \) represents a path from A to D in the original NDFSA.
### Regular expressions

Can write regular language as an expression:

\[ 0^*11^*(0|100^*1)1^*|0^*11^*1 \]

Operators:
- Concatenation (adjacency)
- Or (\( \lor \) or sometime written as \( \lor \))
- Kleene closure (\( ^* \) - 0 or more instances)

### Regular grammars

A regular grammar is a context free grammar where every production is of one of the two forms:

- \( \rightarrow X \rightarrow aY \)
- \( \rightarrow X \rightarrow a \)

for \( X, Y \in N \), \( a \in T \)

**Theorem:** \( L(G) \) for regular grammar \( G \) is equivalent to \( T(M) \) for FSA \( M \).

The proof is “constructive.” That is given either \( G \) or \( M \), can construct the other. [Next slide]
Equivalence of FSA and regular grammars

Extended BNF

This is a shorthand notation for BNF rules. It adds no power to the syntax, only a shorthand way to write productions:

| - Choice
( ) - Grouping
{}* - Repetition - 0 or more
{}+ - Repetition - 1 or more
[] - Optional

Example: Identifier - a letter followed by 0 or more letters or digits:

Extended BNF                      Regular BNF
I \rightarrow L \{ L | D \}^*     I \rightarrow L | L M
L \rightarrow a | b |...         M \rightarrow CM | C
D \rightarrow 0 | 1 |...         C \rightarrow L | D
L \rightarrow a | b |...         L \rightarrow a | b |...
D \rightarrow 0 | 1 |...         D \rightarrow 0 | 1 |...
Syntax diagrams

Also called railroad charts since they look like railroad switching yards.

Trace a path through network: An L followed by repeated loops through L and D, i.e., extended BNF:

$L \rightarrow L (L \mid D)^*$

Syntax charts for expression grammar
Pushdown Automaton (PDA)

A pushdown automaton (PDA) is an abstract machine similar to the DFA

- Has a finite set of states
- Also has a pushdown stack

Moves of the PDA are as follows:

- An input symbol is read and the top symbol on the stack is read
- Based on both inputs, the machine
  - Enters a new state, and
  - Writes zero or more symbols onto the pushdown stack
- String accepted if the stack is empty at end of string

Power of PDAs

PDAs are more powerful than DFAs

- $a^n b^n$, which cannot be recognized by a DFA, can easily be recognized by the PDA
  - Stack all $a$ symbols and, for each $b$, pop an $a$ off the stack.
  - If the end of input is reached at the same time that the stack becomes empty, the string is accepted

As with NFA, we can also have a NDPDA

- NDPDA are more powerful than DPDA
- NDPDA can recognize even length palindromes over $\{0,1\}^*$, but a DPDA cannot. Why? (Hint: Consider palindromes over $\{0,1\}^*2\{0,1\}^*$)

It is true, but less clear, that the languages accepted by NDPDAs are equivalent to the context-free languages

More about these later …
Why do we care about regular languages?

Programs are composed of tokens:
- Identifier
- Number
- Keyword
- Special symbols

Each of these can be defined by regular grammars. (See next slide.)

Problem: How do we handle multiple symbol operators (e.g., ++ in C, += in C, := in Pascal)?
- multiple final states?

Sample token classes
FSA summary

- Scanner for a language turns out to be a giant NDFSA for grammar (i.e., have $\lambda$-rules going from start state to the start state of each token-type on previous slide).