

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

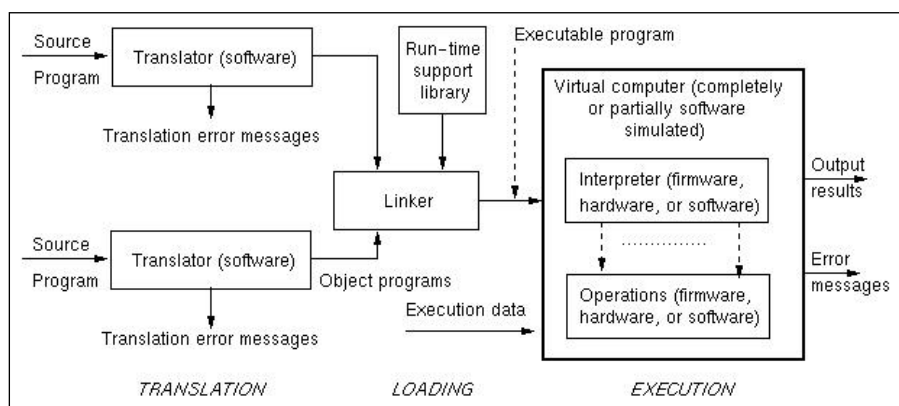
Syntax

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

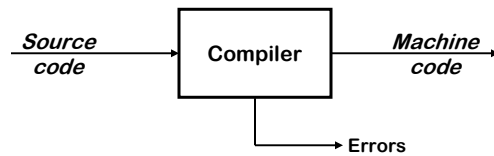
Semantics

- 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



High-level View of a Compiler



Implications

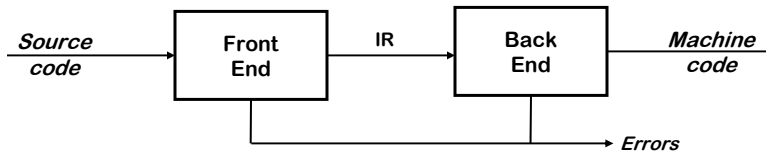
- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations

CMSC430 Spring 2007

9

Traditional Two-pass Compiler



Implications

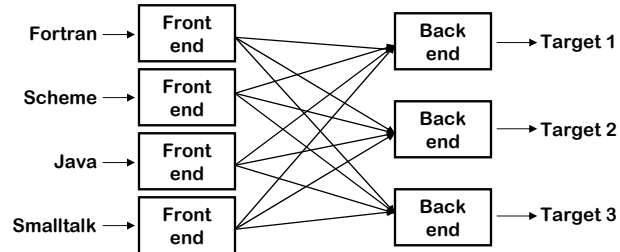
- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes *(better code)*

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

CMSC430 Spring 2007

10

A Common Fallacy



Can we build $n \times m$ compilers with $n+m$ components?

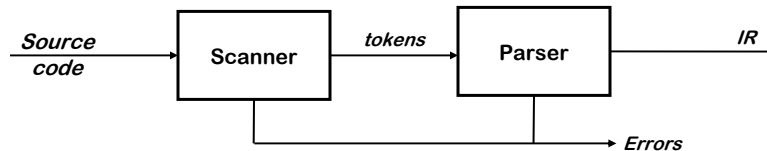
- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Limited success in systems with very low-level IRs

CMSC430 Spring 2007

11

The Front End



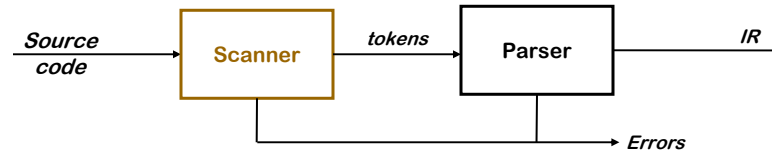
Responsibilities

- Recognize legal (& 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

CMSC430 Spring 2007

12

The Front End



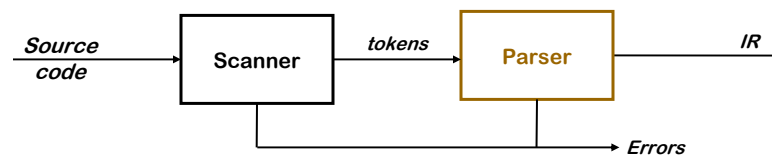
Scanner

- 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* \cong *lexeme*, *part of speech* \cong *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

CMSC430 Spring 2007

13

The Front End



Parser

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

Hand-coded parsers are fairly easy to build
Most books advocate using automatic parser generators

CMSC430 Spring 2007

14

The Front End

Context-free syntax is specified with a grammar

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \textit{SheepNoise} \underline{\textit{baa}} \\ &| \underline{\textit{baa}} \end{aligned}$$

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* ($P: N \rightarrow N \cup T$)

Example due to Dr. Scott K. Warren

The Front End

Context-free syntax can be put to better use

1. $\textit{goal} \rightarrow \textit{expr}$
2. $\textit{expr} \rightarrow \textit{expr} \textit{op} \textit{term}$
3. \textit{term}
4. $\textit{term} \rightarrow \underline{\textit{number}}$
5. $\underline{\textit{id}}$
6. $\textit{op} \rightarrow +$
7. $-$

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

- 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

Production	Result
	<i>goal</i>
1	<i>expr</i>
2	<i>expr op term</i>
5	<i>expr op y</i>
7	<i>expr - y</i>
2	<i>expr op term - y</i>
4	<i>expr op 2 - y</i>
6	<i>expr + 2 - y</i>
3	<i>term + 2 - y</i>
5	<i>x + 2 - y</i>

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

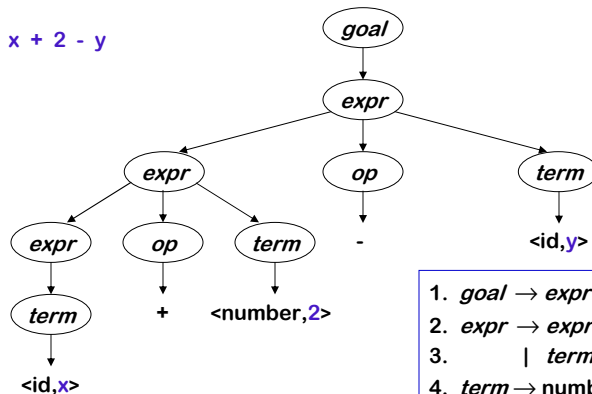
CMSC430 Spring 2007

17

The Front End

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

$x + 2 - y$



This contains a lot of unneeded Information.

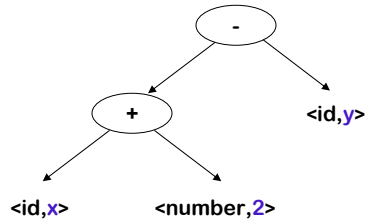
1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → *number*
5. | *id*
6. *op* → +
7. | -

CMSC430 Spring 2007

18

The Front End

Compilers often use an *abstract syntax tree*

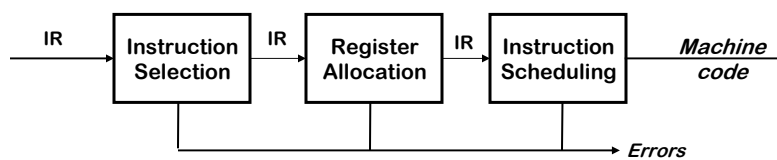


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

ASTs are one form of *intermediate representation (IR)*

The Back End

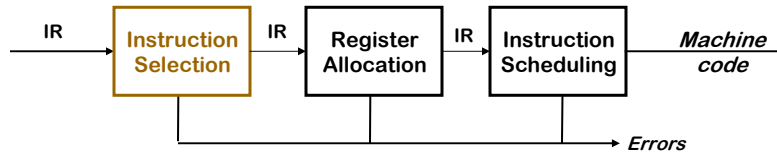


Responsibilities

- 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



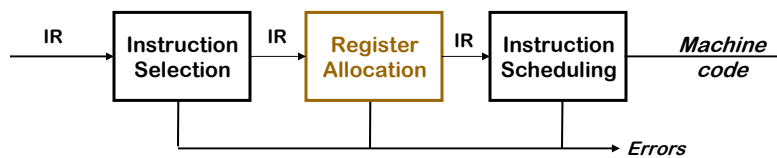
Instruction Selection

- 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

The Back End

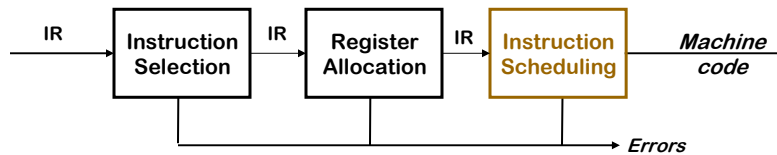


Register allocation

- 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)

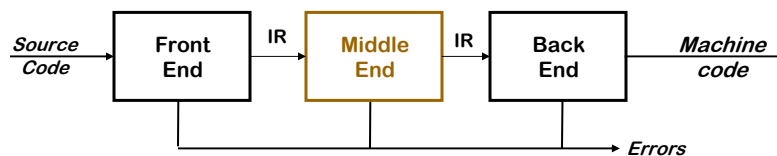
Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

CMSC430 Spring 2007

23

Traditional Three-pass Compiler



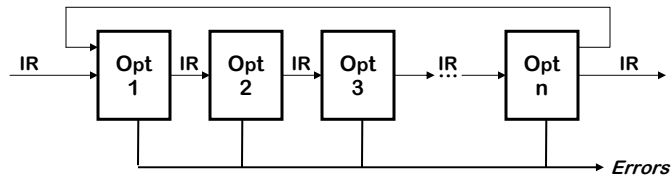
Code Improvement (or *Optimization*)

- 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

CMSC430 Spring 2007

24

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?]

$$\text{Address}(A(I,J)) = \text{address}(A(0,0)) + J * (\text{column size}) + I$$

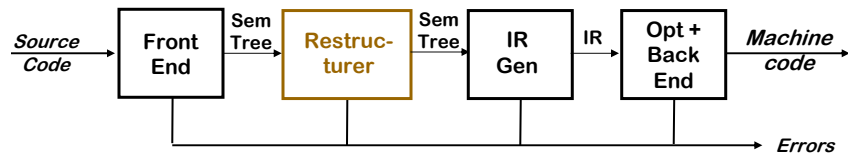
Does the user realize a multiplication is generated here?

```
DO I = 1, M
  A(I,J) = A(I,J) + C
ENDDO
```



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

Modern Restructuring Compiler



Typical **Restructuring** Transformations:

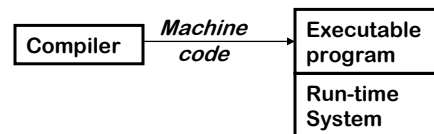
- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining

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.

CMSC430 Spring 2007

29

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 \Rightarrow)

CMSC430 Spring 2007

30

Example BNF sentences

<sentence> \Rightarrow <subject> <predicate> **First rule**

\Rightarrow <article> <noun> <predicate> **Second rule**

\Rightarrow the <noun> <predicate> **Fifth rule**

... \Rightarrow the boy ate the cake

Also from <sentence> you can derive

\Rightarrow the cake ate the boy

Syntax does not imply correct semantics

Note:

Rule <A> ::= <C>

This BNF rule also written with equivalent syntax:

$A \rightarrow BC$

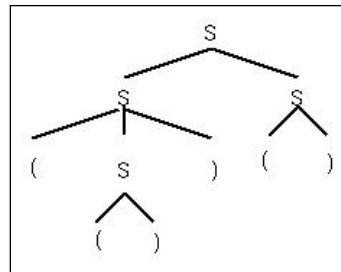
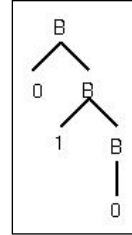
Languages

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
 - > $S \rightarrow SS \mid (S) \mid ()$
 - > $S \Rightarrow SS \Rightarrow (S)S \Rightarrow (()S \Rightarrow (()()$
 - > $S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (()()$
- Different derivations but get the **same** parse tree



CMSC430 Spring 2007

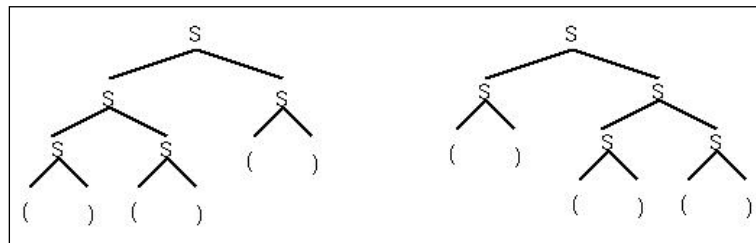
33

Ambiguity

But from some grammars you can get 2 **different** parse trees for the same string: $()()$

- Each corresponds to a unique derivation:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()$$



- A grammar is **ambiguous** if some sentence has 2 distinct parse trees.
- We desire unambiguous grammars to understand semantics.

CMSC430 Spring 2007

34

Role of λ (or ϵ)

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

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

Can always delete them in grammar. For example:

$X \rightarrow abYc$

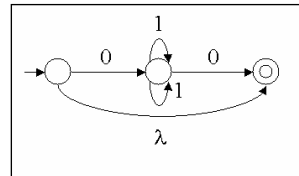
$Y \rightarrow \epsilon$

Delete ϵ -production and add production without ϵ :

$X \rightarrow abYc$

$X \rightarrow abc$

2. In FSA - λ moves means that in initial state, without input you can move to final state.



CMSC430 Spring 2007

35

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?**

CMSC430 Spring 2007

36

Usual grammar for expressions

$E \rightarrow E + T \mid T$

$T \rightarrow T * P \mid P$

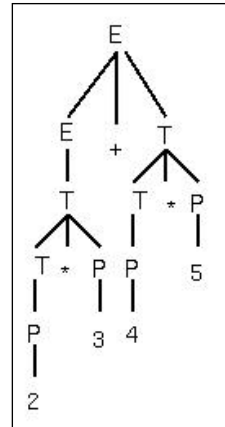
$P \rightarrow i \mid (E)$

“Natural” value of expression
is 26

Multiply $2 * 3 = 6$

Multiply $4 * 5 = 20$

Add $6 + 20 = 26$



But the “precedence” of operations is only a convention

Grammar for 70

$E \rightarrow E * T \mid T$

$T \rightarrow T + P \mid P$

$P \rightarrow i \mid (E)$

Grammar for 50

$E \rightarrow E + T \mid E * T \mid T$

$T \rightarrow i \mid (E)$

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 $2*3+4*5$ 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

1	<i>Goal</i>	→	<i>Expr</i>
2	<i>Expr</i>	→	<i>Expr Op Expr</i>
3			<u>number</u>
4			<u>id</u>
5	<i>Op</i>	→	+
6			-
7			*
8			/

<i>Rule</i>	<i>Sentential Form</i>
—	<i>Expr</i>
2	<i>Expr Op Expr</i>
4	<id, <u>x</u> > <i>Op Expr</i>
6	<id, <u>x</u> > - <i>Expr</i>
2	<id, <u>x</u> > - <i>Expr Op Expr</i>
3	<id, <u>x</u> > - <num, <u>2</u> > <i>Op Expr</i>
7	<id, <u>x</u> > - <num, <u>2</u> > * <i>Expr</i>
4	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >

We denote this: $Expr \Rightarrow^* \underline{id} - \underline{num} * \underline{id}$

- Such a sequence of rewrites is called a *derivation*
- Process of discovering a derivation is called *parsing*

The Two Derivations for $x - 2 * y$

Rule	Sentential Form
—	<i>Expr</i>
2	<i>Expr Op Expr</i>
4	$\langle \text{id}, x \rangle \text{ Op Expr}$
6	$\langle \text{id}, x \rangle - \text{Expr}$
2	$\langle \text{id}, x \rangle - \text{Expr Op Expr}$
3	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle \text{ Op Expr}$
7	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \text{Expr}$
4	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

Leftmost derivation

Rule	Sentential Form
—	<i>Expr</i>
2	<i>Expr Op Expr</i>
4	<i>Expr Op</i> $\langle \text{id}, y \rangle$
7	<i>Expr</i> * $\langle \text{id}, y \rangle$
2	<i>Expr Op Expr</i> * $\langle \text{id}, y \rangle$
3	<i>Expr Op</i> $\langle \text{num}, 2 \rangle$ * $\langle \text{id}, y \rangle$
6	<i>Expr</i> - $\langle \text{num}, 2 \rangle$ * $\langle \text{id}, y \rangle$
4	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

Rightmost derivation

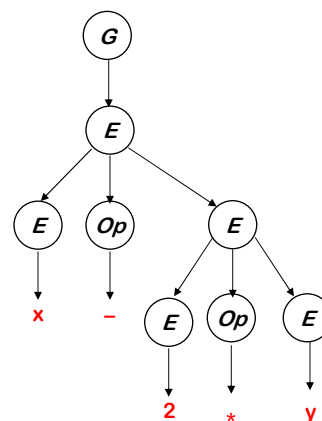
In both cases, $\text{Expr} \Rightarrow^* \text{id} - \text{num} * \text{id}$

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

Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	<i>Expr</i>
2	<i>Expr Op Expr</i>
4	$\langle \text{id}, x \rangle \text{ Op Expr}$
6	$\langle \text{id}, x \rangle - \text{Expr}$
2	$\langle \text{id}, x \rangle - \text{Expr Op Expr}$
3	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle \text{ Op Expr}$
7	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \text{Expr}$
4	$\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

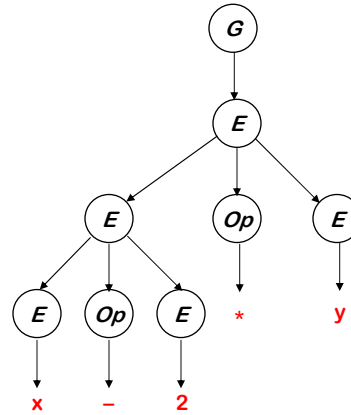


This evaluates as $x - (2 * y)$

Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	$Expr$
2	$Expr Op Expr$
4	$Expr Op \langle id, y \rangle$
7	$Expr * \langle id, y \rangle$
2	$Expr Op Expr * \langle id, y \rangle$
3	$Expr Op \langle num, 2 \rangle * \langle id, y \rangle$
6	$Expr - \langle num, 2 \rangle * \langle id, y \rangle$
4	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$



This evaluates as $(x - 2) * y$

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:

1	<i>Goal</i>	\rightarrow	<i>Expr</i>
2	<i>Expr</i>	\rightarrow	<i>Expr</i> + <i>Term</i>
3			<i>Expr</i> - <i>Term</i>
4			<i>Term</i>
5	<i>Term</i>	\rightarrow	<i>Term</i> * <i>Factor</i>
6			<i>Term</i> / <i>Factor</i>
7			<i>Factor</i>
8	<i>Factor</i>	\rightarrow	<u>number</u>
9			<u>id</u>
10			(<i>Expr</i>)

This grammar is slightly larger

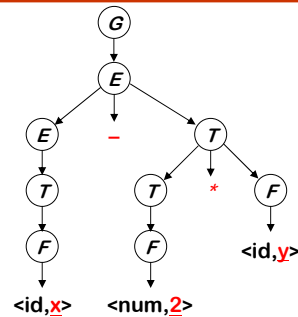
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses our example

Derivations and Precedence

Rule	Sentential Form
—	<i>Expr</i>
3	<i>Expr</i> - <i>Term</i>
5	<i>Expr</i> - <i>Term</i> * <i>Factor</i>
9	<i>Expr</i> - <i>Term</i> * <id,y>
7	<i>Expr</i> - <i>Factor</i> * <id,y>
8	<i>Expr</i> - <num,2>* <id,y>
4	<i>Term</i> - <num,2>* <id,y>
7	<i>Factor</i> - <num,2>* <id,y>
9	<id,x> - <num,2>* <id,y>

The rightmost derivation



Its parse tree

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

1	<i>Goal</i>	→	<i>Expr</i>
2	<i>Expr</i>	→	<i>Expr Op Expr</i>
3			<u>number</u>
4			<u>id</u>
5	<i>Op</i>	→	+
6			-
7			*
8			/

Rule	Sentential Form
—	<i>Expr</i>
2	<i>Expr Op Expr</i>
②	<i>Expr Op Expr Op Expr</i>
4	<id, <i>x</i> > <i>Op Expr Op Expr</i>
6	<id, <i>x</i> > - <i>Expr Op Expr</i>
3	<id, <i>x</i> > - <num, <i>2</i> > <i>Op Expr</i>
7	<id, <i>x</i> > - <num, <i>2</i> > * <i>Expr</i>
4	<id, <i>x</i> > - <num, <i>2</i> > * <id, <i>y</i> >

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

different choice
than the first time

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is *ambiguous*
- If a grammar has more than one rightmost derivation for a single *sentential form*, the grammar is *ambiguous*
- The leftmost and rightmost derivations for a *sentential form* may differ, even in an unambiguous grammar

Classic example — the *if-then-else* problem

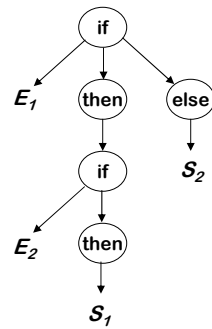
Stmt → if *Expr* then *Stmt*
 | if *Expr* then *Stmt* else *Stmt*
 | ... *other stmts* ...

This ambiguity is entirely grammatical in nature

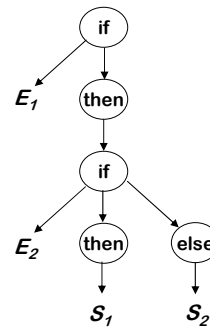
Ambiguity

This sentential form has two derivations

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$



production 2, then
production 1



production 1, then
production 2

CMSC430 Spring 2007

49

Ambiguity

Removing the ambiguity

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

1	$Stmt \rightarrow WithElse$
2	$NoElse$
3	$WithElse \rightarrow \underline{if} Expr \underline{then} WithElse \underline{else} WithElse$
4	$OtherStmt$
5	$NoElse \rightarrow \underline{if} Expr \underline{then} Stmt$
6	$\underline{if} Expr \underline{then} WithElse \underline{else} NoElse$

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

50

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*)

```
1 | Stmt → WithElse
2 |       | NoElse
3 | WithElse → if Expr then WithElse else Stmt
4 |       | OtherStmt
5 | NoElse → if Expr then Stmt
```

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

With this grammar, the example has only one derivation

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

CMSC430 Spring 2007

53

Classes of grammars

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

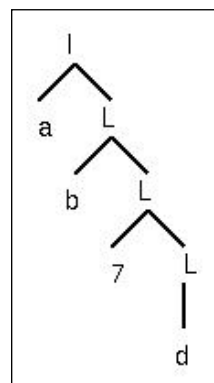
Regular grammars: subclass of BNF - Type 3:

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:
 $I \rightarrow a L \mid b L \mid c L \mid \dots \mid z L \mid a \mid \dots \mid y \mid z$
 $L \rightarrow 1 L \mid 2 L \mid \dots \mid 9 L \mid 0 L \mid 1 \mid \dots \mid 9 \mid 0 \mid a L \mid b L \mid c L$
 $\mid \dots \mid z L \mid a \mid \dots \mid y \mid z$



ab7d

CMSC430 Spring 2007

54

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 $\alpha \leq$ length of β , i.e., all sentential forms are length non-decreasing]

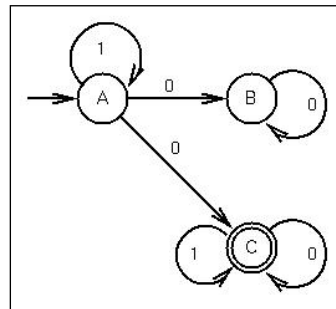
Unrestricted, recursively enumerable: Type 0 - Rules: $\alpha \rightarrow \beta$. No restrictions on α and β .

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**.

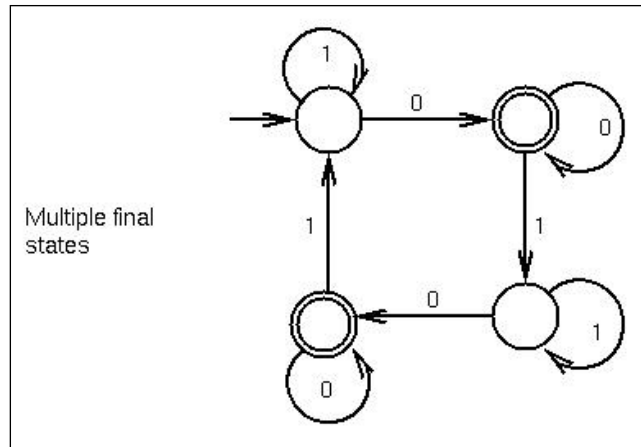
What strings, starting in state A, end up at state C?



The language accepted by machine M is set of strings that move from start node to a final node, or more formally: $T(M) = \{\omega \mid \delta(A, \omega) = C\}$ where A is start node and C a final node.

More on FSAs

An FSA can have more than one final state:



CMSC430 Spring 2007

57

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):

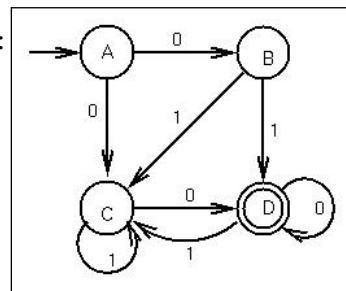
01 is accepted via path:

ABD

even though 01 also can take the paths:

ACC or ABC

and C is not a final state.



CMSC430 Spring 2007

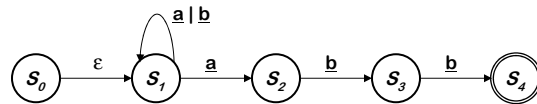
58

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 $(a | b)^* abb$?



This is a little different

- s_0 has a transition on ϵ
- s_1 has two transitions on a

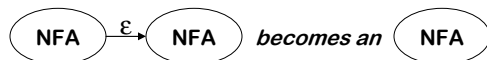
This is a *non-deterministic finite automaton* (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string x iff \exists a path through the transition graph from s_0 to a final state such that the edge labels spell x
- Transitions on ϵ 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 ϵ -transitions



Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no ϵ 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

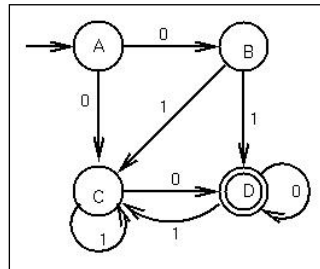
Equivalence of FSA and NDFSA

Important early result:

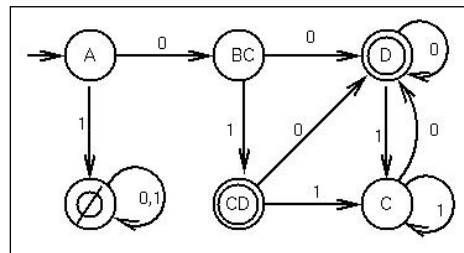
NDFSA = DFSA

Let subsets of states be states in DFSA.

Keep track of which subset you can be in.



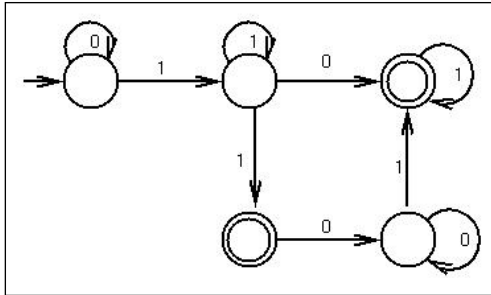
Any string from {A} to either {D} or {CD} represents a path from A to D in the original NDFSA.



Regular expressions

Can write regular language as an expression:

$0^*11^*(0|100^*1)1^*|0^*11^*1$



Operators:

Concatenation (adjacency)

Or (| or sometime written as \vee)

Kleene closure ($*$ - 0 or more instances)

CMSC430 Spring 2007

63

Regular grammars

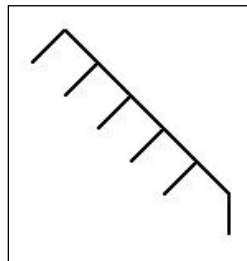
A regular grammar is a context free grammar where every production is of

one of the two forms:

$\rightarrow X \rightarrow aY$

$\rightarrow X \rightarrow a$

for $X, Y \in N, a \in T$



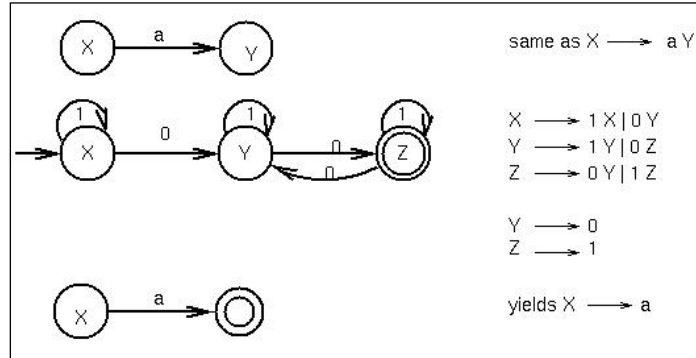
Theorem: $L(G)$ for regular grammar G is equivalent to $T(M)$ for FSA M .

The proof is “constructive.” That is given either G or M , can construct the other. [Next slide]

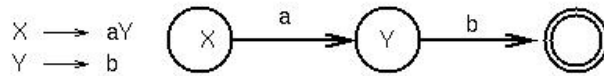
CMSC430 Spring 2007

64

Equivalence of FSA and regular grammars



To go from regular grammar to FSA, make the following transformations:



CMSC430 Spring 2007

65

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

$I \rightarrow L \{ L \mid D \}^*$

$L \rightarrow a \mid b \mid \dots$

$D \rightarrow 0 \mid 1 \mid \dots$

Regular BNF

$I \rightarrow L \mid LM$

$M \rightarrow CM \mid C$

$C \rightarrow L \mid D$

$L \rightarrow a \mid b \mid \dots$

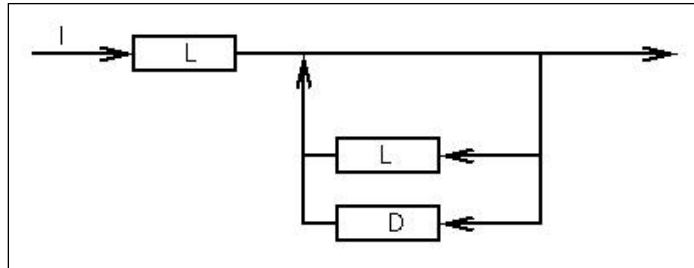
$D \rightarrow 0 \mid 1 \mid \dots$

CMSC430 Spring 2007

66

Syntax diagrams

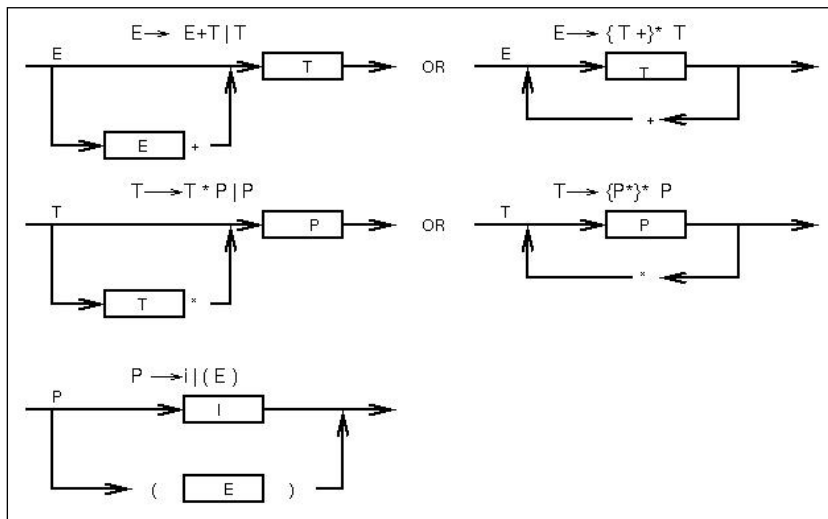
Also called railroad charts since they look like railroad switching yards.



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

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

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.)

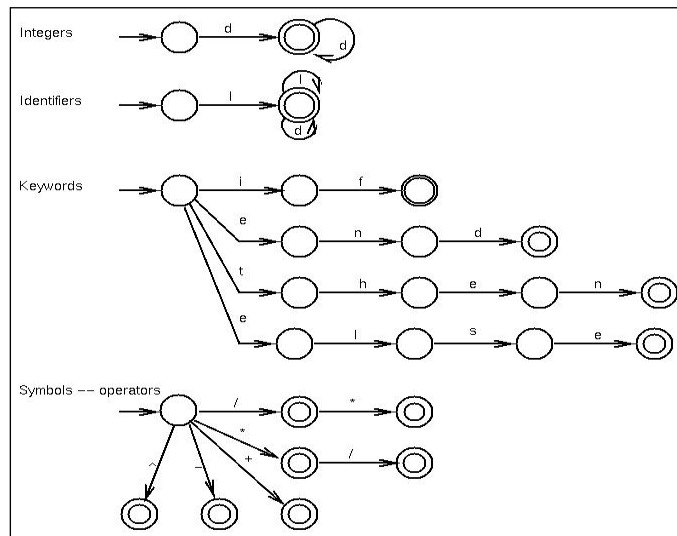
Problem: How do we handle multiple symbol operators (e.g., ++ in C, += in C, := in Pascal)?

?? -multiple final states?

CMSC430 Spring 2007

71

Sample token classes



CMSC430 Spring 2007

72

FSA summary

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

