

www.vuCTn.com
CS606 Complete MERGED
PPT Slides 1 TO 45

Regards : Zarva Chaudhary

Admin:

* Zarva Chaudhary *

* Laiba Maki *

Merged File date

31 -07 -2021

Chaudhary Moazzam

Compiler Construction

CS 606
Sohail Aslam
Lecture 1

Course Organization

- General course information
- Homework and project information

2

General Information

Instructor	Sohail Aslam
Lectures	45
Text	<i>Compilers – Principles, Techniques and Tools</i> by Aho, Sethi and Ullman

3

Work Distribution

- Theory

- homeworks = 20%
- exams = 40%

- Practice

- build a compiler = 40%

4

Project

Implementation

- language: subset of java
- Generated code: Intel x86 assembly
- Implementation language: C++
- Six programming assignments

5

Why Take this Course

Reason #1: understand compilers and languages

- understand the code structure
- understand language semantics
- understand relation between source code and generated machine code
- become a better programmer

6

Why Take this Course

Reason #2: nice balance of theory and practice

- Theory
 - mathematical models: regular expressions, automata, grammars, graphs
 - algorithms that use these models

7

Why Take this Course

Reason #2: nice balance of theory and practice

- Practice

- Apply theoretical notions to build a real compiler

8

Why Take this Course

Reason #3: programming experience

- write a large program which manipulates complex data structures
- learn more about C++ and Intel x86

9

What are Compilers

- Translate information from one representation to another
- Usually **information = program**

10

Examples

- Typical Compilers:
 - VC, VC++, GCC, JavaC
 - FORTRAN, Pascal, VB(?)
- Translators
 - Word to PDF
 - PDF to Postscript

11

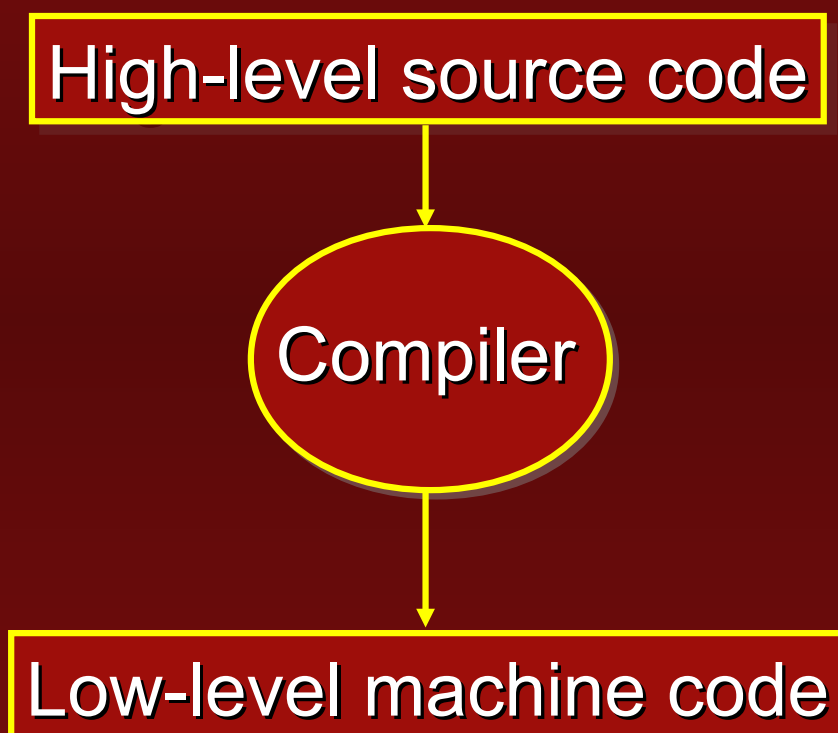
In This Course

We will study typical compilation:

- from programs written in high-level languages to low-level object code and machine code

12

Typical Compilation



13

Source Code

```
int expr( int n )  
{  
    int d;  
    d = 4*n*n* (n+1) * (n+1) ;  
    return d;  
}
```

14

Source Code

- Optimized for human readability
- Matches human notions of grammar
- Uses named constructs such as variables and procedures

15

Assembly Code

```
.globl _expr
_expr:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl 8(%ebp),%eax
    movl %eax,%edx
    leal 0(,%edx,4),%eax
    movl %eax,%edx
    imull 8(%ebp),%edx
    movl 8(%ebp),%eax
    incl %eax
    imull %eax,%edx
    movl 8(%ebp),%eax
    incl %eax
    imull %eax,%edx
    movl %edx,-4(%ebp)
    movl -4(%ebp),%edx
    movl %edx,%eax
    jmp L2
    .align 4
L2:
    leave
    ret
```

16

Assembly Code

Optimized for hardware

- Consists of machine instructions
- Uses registers and unnamed memory locations
- Much harder to understand by humans

17

How to Translate

Correctness:

the generated machine code must execute precisely the same computation as the source code

18

How to Translate

- Is there a unique translation?
No!
- Is there an algorithm for an “ideal translation”? **No!**

19

How to Translate

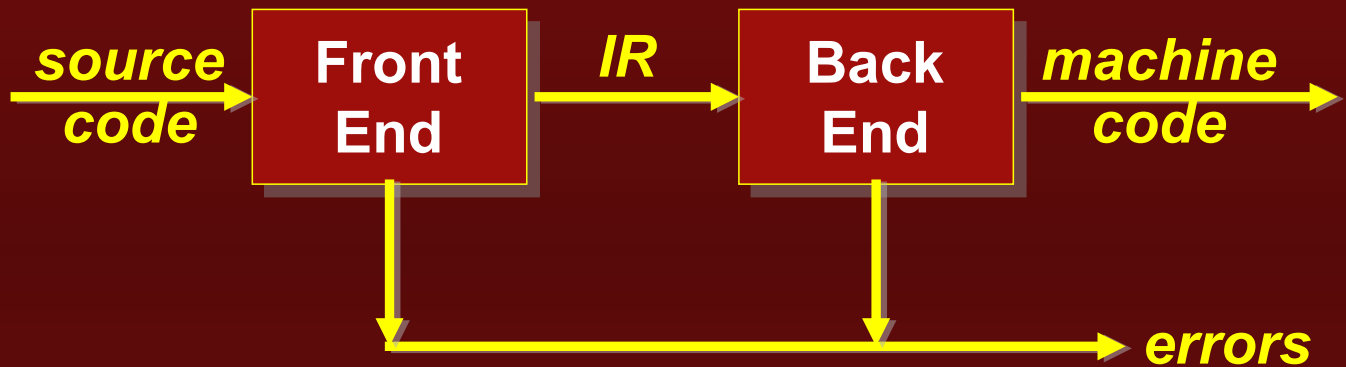
- Translation is a complex process
- source language and generated code are very different
- Need to structure the translation

20

Compiler Construction

CS 606
Sohail Aslam
Lecture 2

Two-pass Compiler



2

Two-pass Compiler

- Use an intermediate representation (IR)
- **Front end** maps legal source code into IR

3

Two-pass Compiler

- **Back end** maps IR into target machine code
- Admits multiple front ends & multiple passes

4

Two-pass Compiler

- Front end is $O(n)$ or $O(n \log n)$
- Back end is ***NP-Complete*** (NPC)

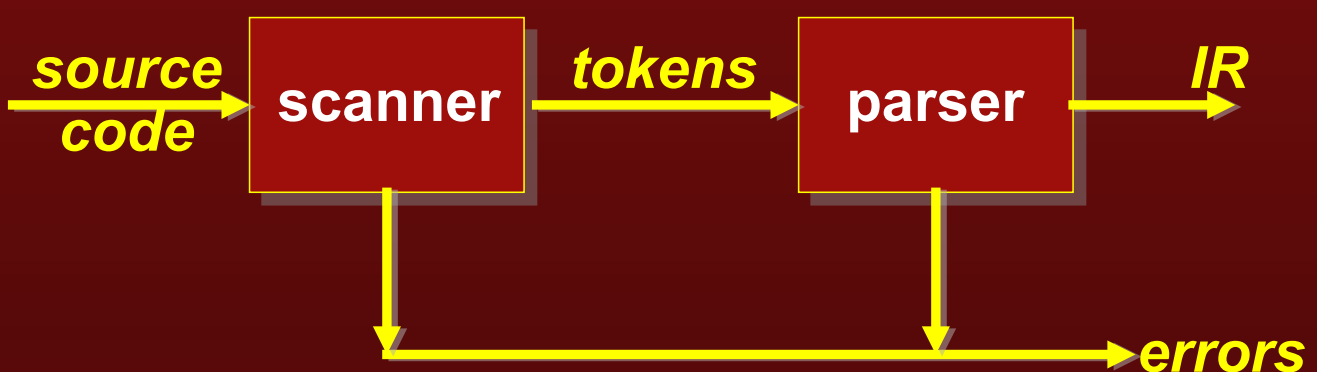
5

Front End

- Recognizes legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map

6

The Front-End

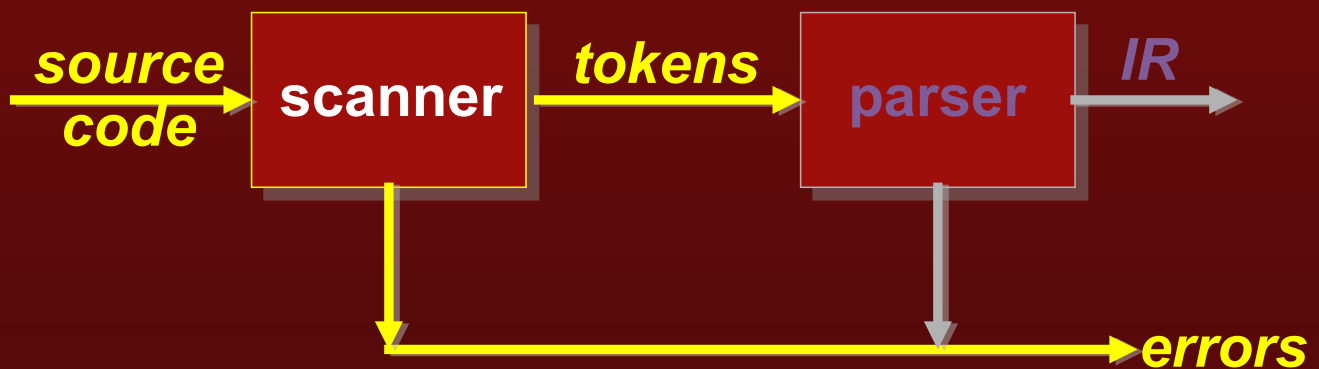


Modules

- Scanner
- Parser

7

Scanner



8

Scanner

- Maps character stream into **words** – basic unit of syntax
- Produces **pairs** –
 - a word and
 - its part of speech

9

Scanner

- Example

x = x + y

becomes

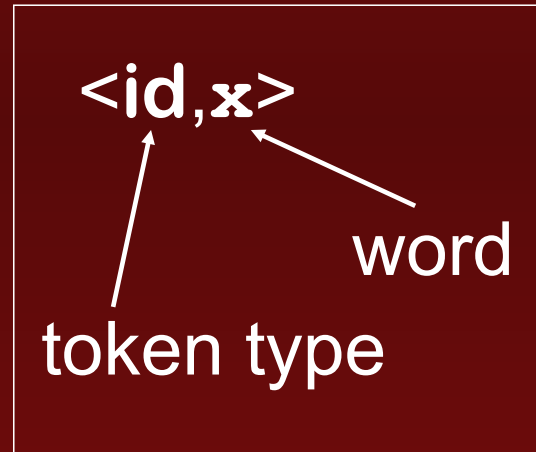
<id,x>

<assign,=>

<id,x>

<op,+>

<id,y>



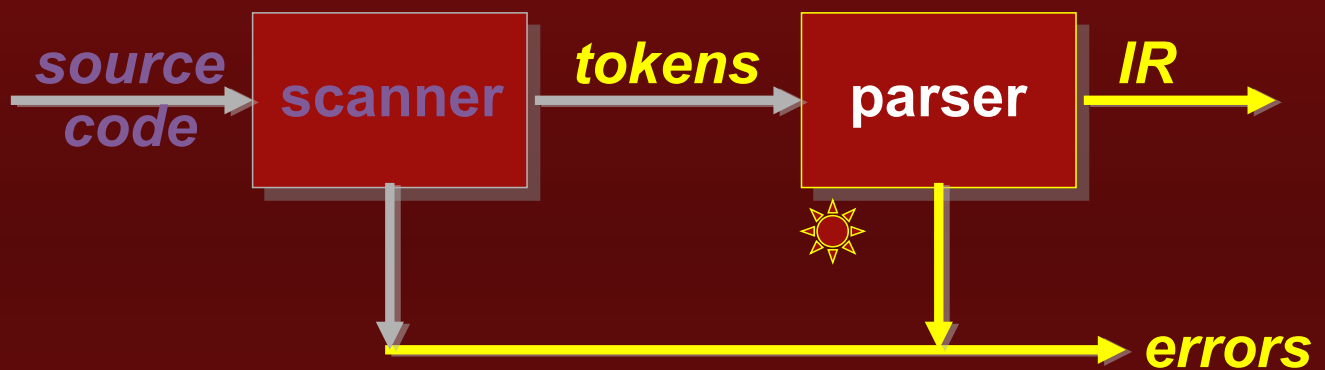
10

Scanner

- we call the pair
“<token type, word>” a
“**token**”
- typical tokens: *number*,
identifier, *+*, *-*, *new*, *while*, *if*

11

Parser



12

Parser

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

13

Context-Free Grammars

- Context-free syntax is specified with a grammar $G=(S,N,T,P)$
- S is the *start* symbol
- N is a set of *non-terminal* symbols
- T is set of *terminal* symbols or words
- P is a set of *productions* or rewrite rules

14

Context-Free Grammars

Grammar for expressions

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

15

The Front End

- For this CFG

$S = \textit{goal}$

$T = \{ \textit{number}, \textit{id}, +, - \}$

$N = \{ \textit{goal}, \textit{expr}, \textit{term}, \textit{op} \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$

16

Context-Free Grammars

- Given a CFG, we can derive sentences by repeated substitution
- Consider the sentence (expression)

$x + 2 - y$

17

Derivation

<u>Production</u>	<u>Result</u>
	<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>

18

The Front End

- To recognize a valid sentence in some CFG, we **reverse** this process and build up a **parse**
- A parse can be represented by a tree: ***parse tree*** or ***syntax tree***

19

Parse

Production

Result

1

goal

2

expr

5

expr op term

7

expr op y

2

expr – y

4

expr op term – y

6

expr op 2 – y

3

expr + 2 – y

5

term + 2 – y

x + 2 – y



20

Compiler Construction

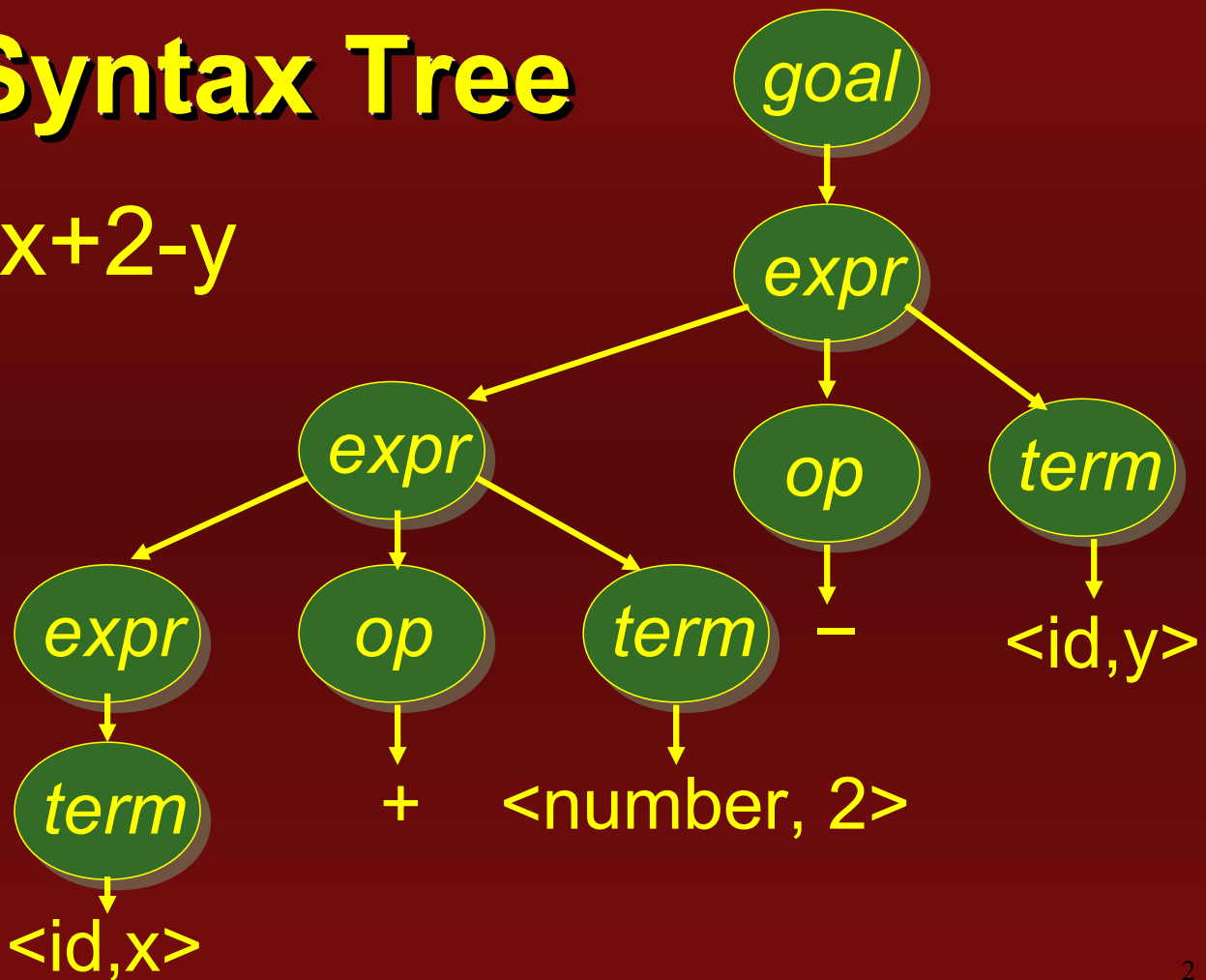
CS 606

Sohail Aslam

Lecture 3

Syntax Tree

$x+2-y$



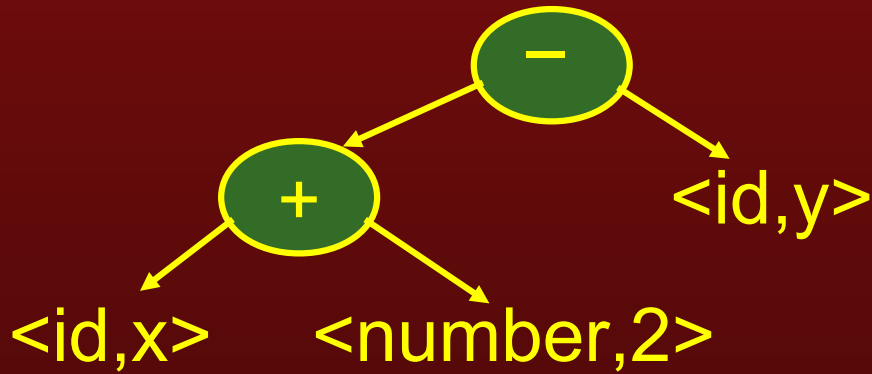
2

Abstract Syntax Trees

- The parse tree contains a lot of unneeded information.
- Compilers often use an *abstract syntax tree (AST)*.

3

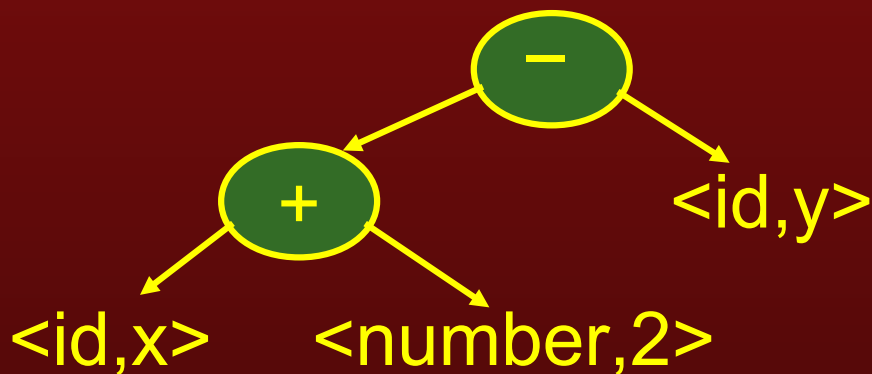
Abstract Syntax Trees



- This is much more concise

4

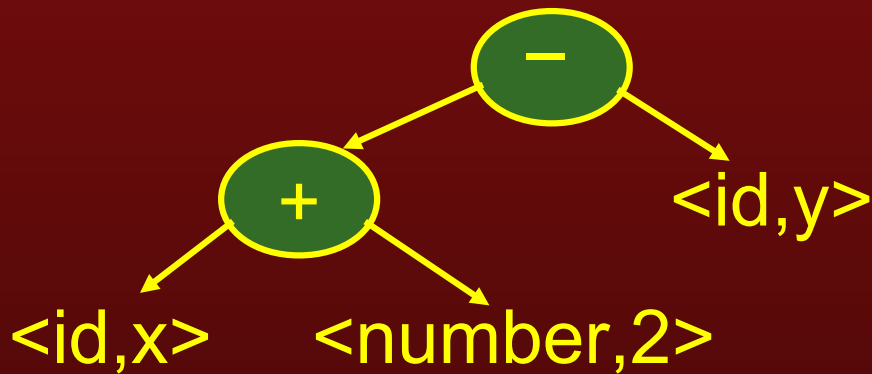
Abstract Syntax Trees



- AST summarizes grammatical structure without the details of derivation

5

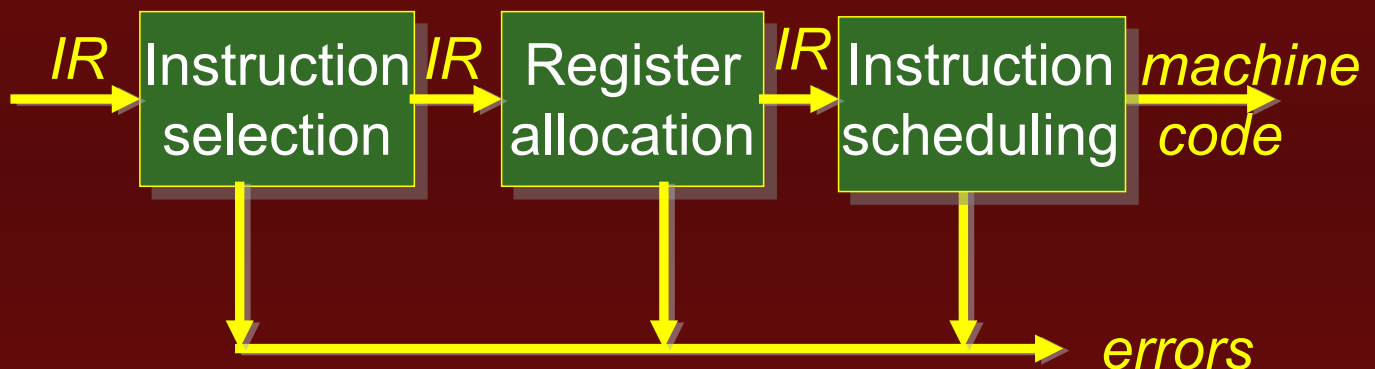
Abstract Syntax Trees



- ASTs are one kind of *intermediate representation (IR)*

6

The Back End



7

The Back End

- Translate IR into target machine code.
- Choose machine (assembly) instructions to implement each IR operation

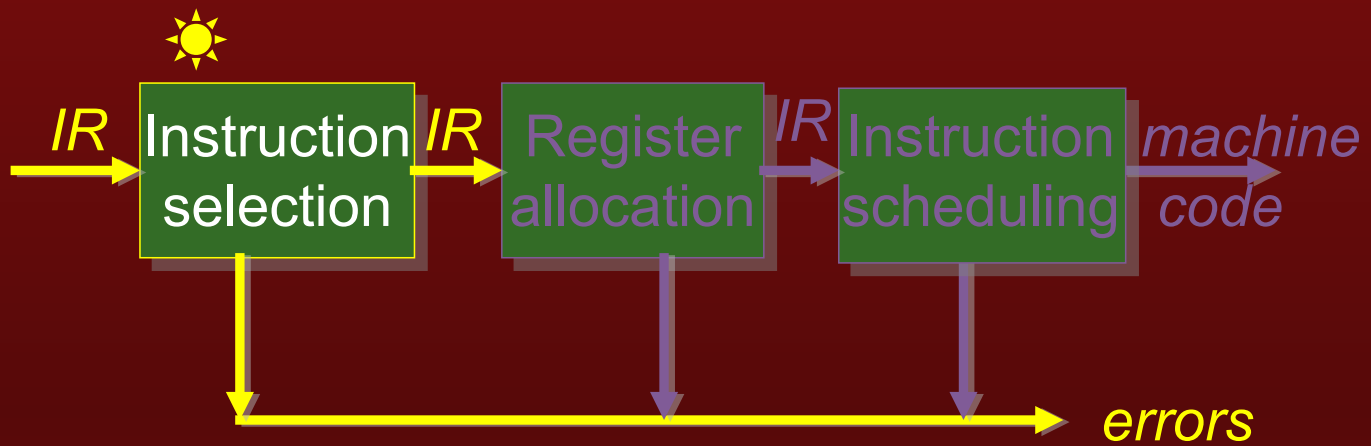
8

The Back End

- Ensure conformance with system interfaces
- Decide which values to keep in registers

9

The Back End

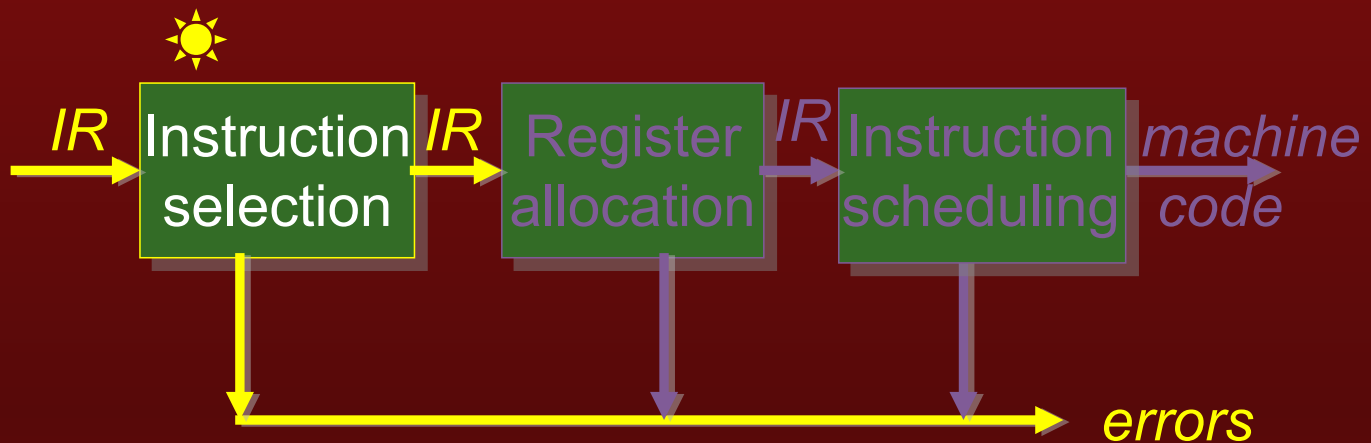


Instruction Selection:

- Produce fast, compact code.

10

The Back End

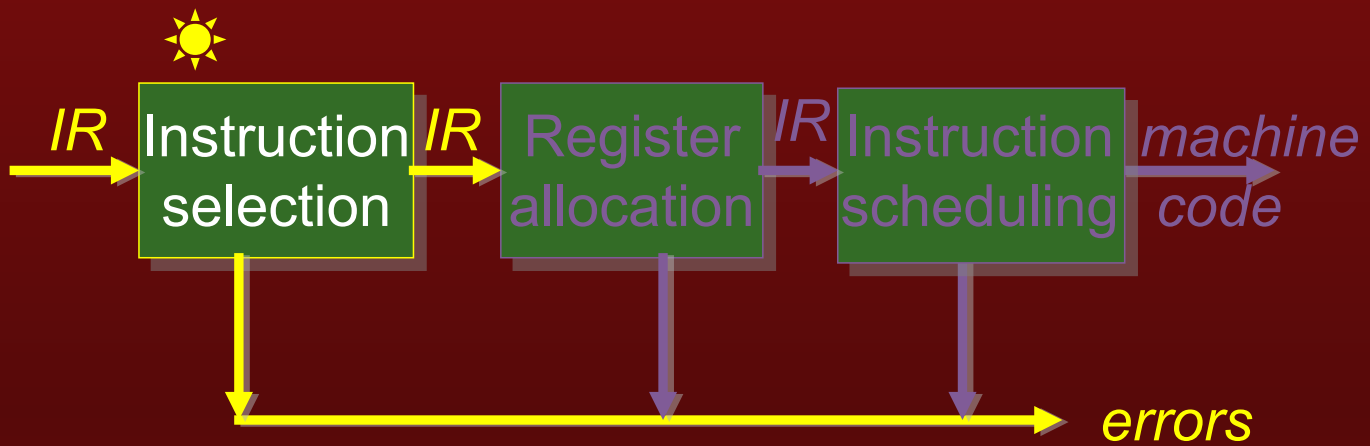


Instruction Selection:

- Take advantage of target features such as addressing modes.

11

The Back End

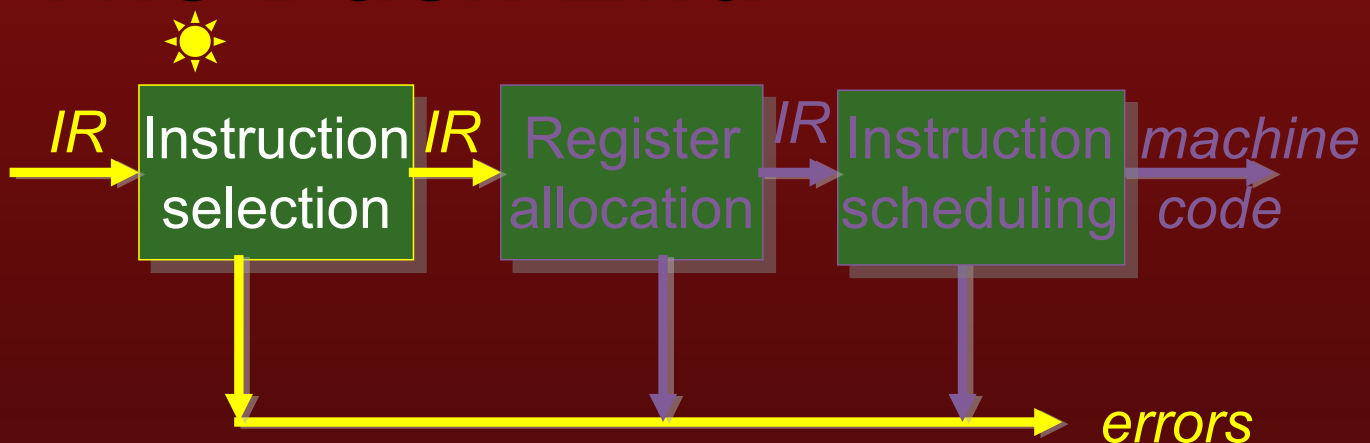


Instruction Selection:

- Usually viewed as a pattern matching problem – dynamic programming.

12

The Back End

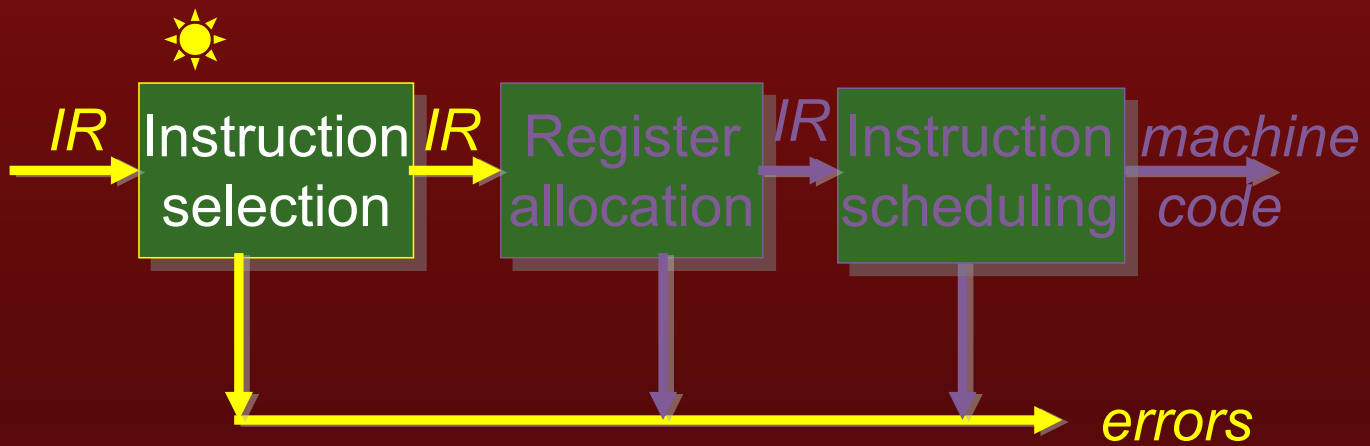


Instruction Selection:

- Spurred by PDP-11 to VAX-11 - CISC.

13

The Back End

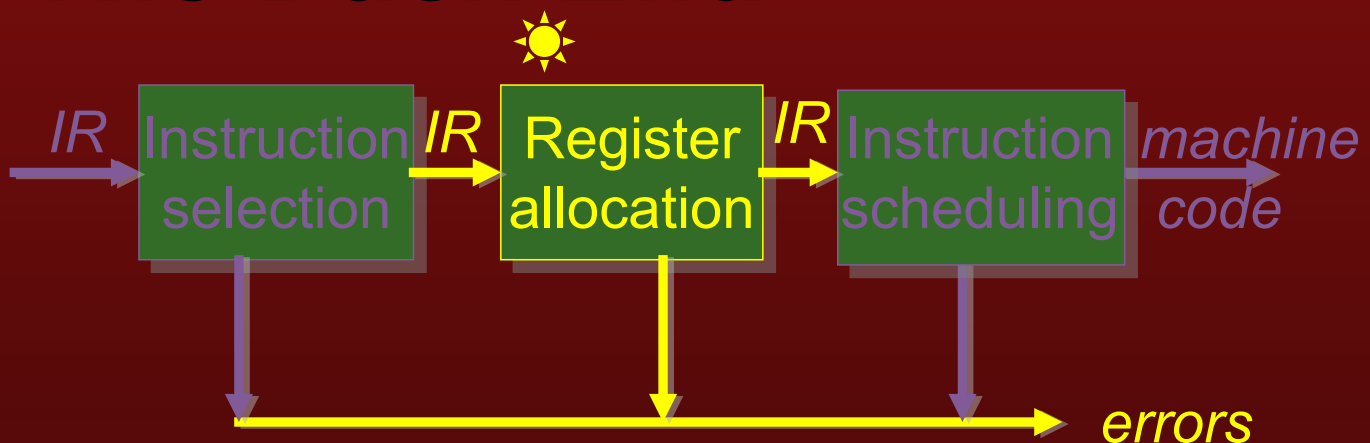


Instruction Selection:

- RISC architecture simplified this problem.

14

The Back End

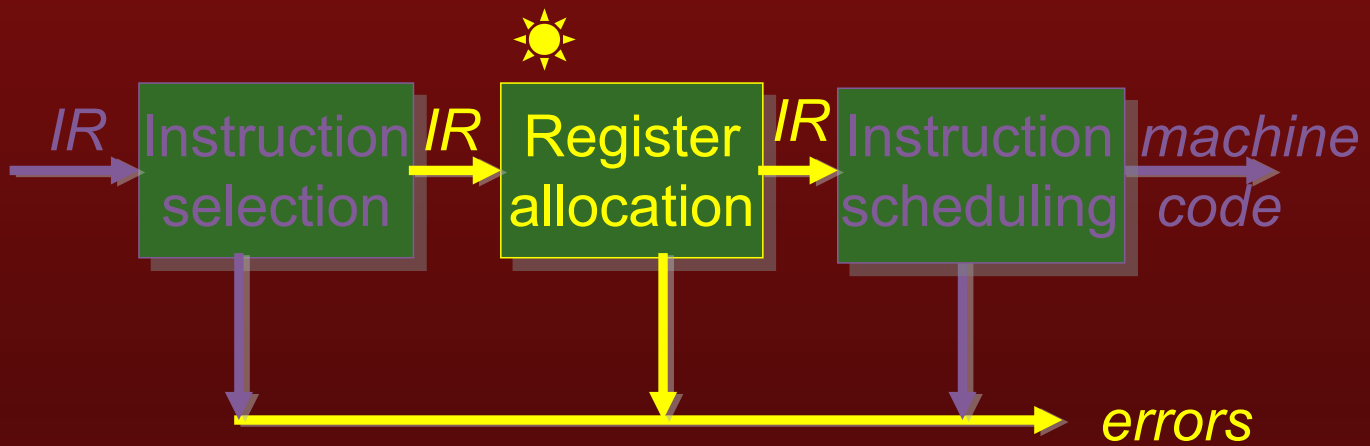


Register Allocation:

- Have each value in a register when it is used.

15

The Back End

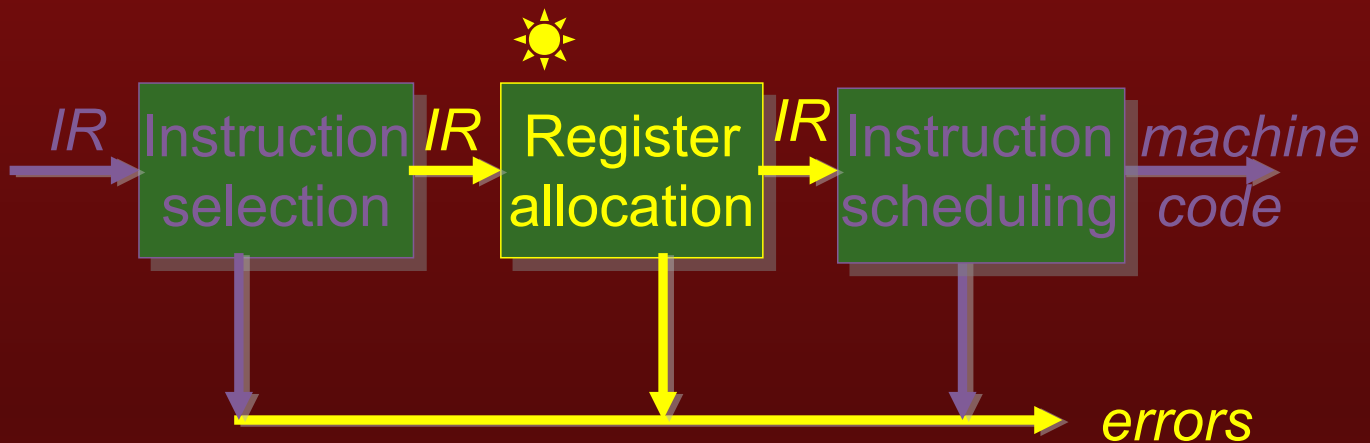


Register Allocation:

- Manage a limited set of resources – register file.

16

The Back End

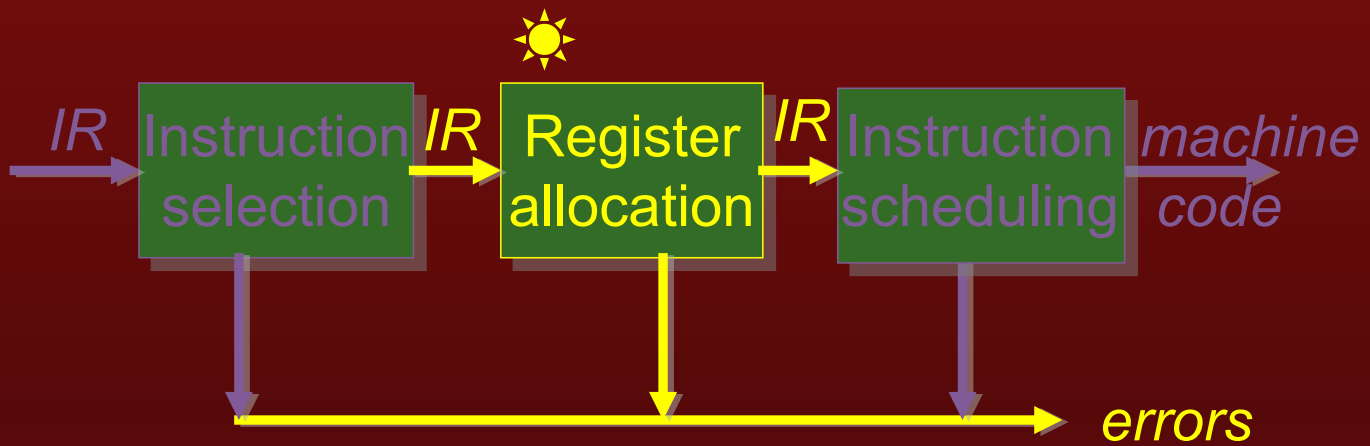


Register Allocation:

- Can change instruction choices and insert LOADs and STOREs.

17

The Back End

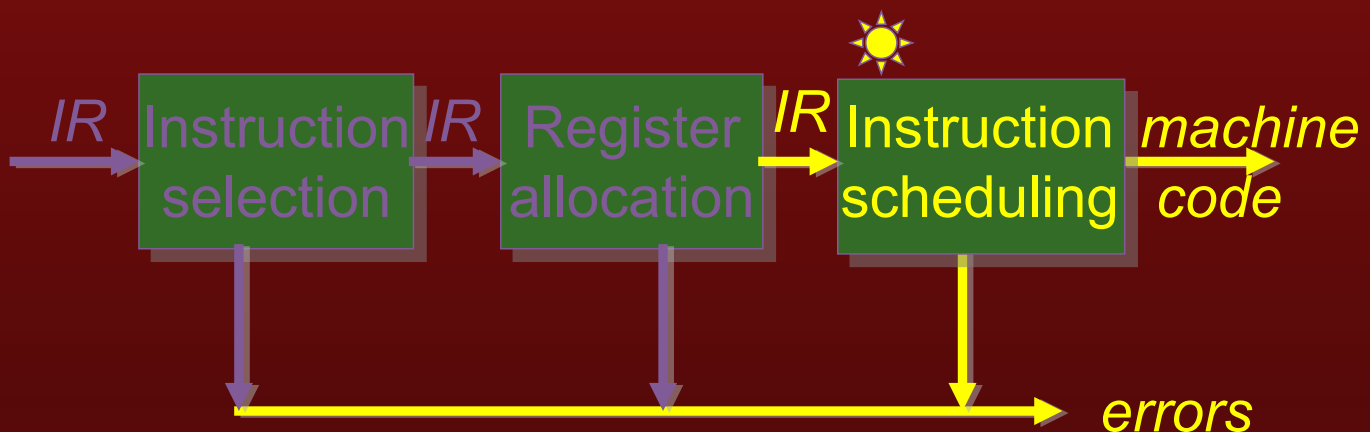


Register Allocation:

- Optimal register allocation is *NP-Complete*.

18

The Back End

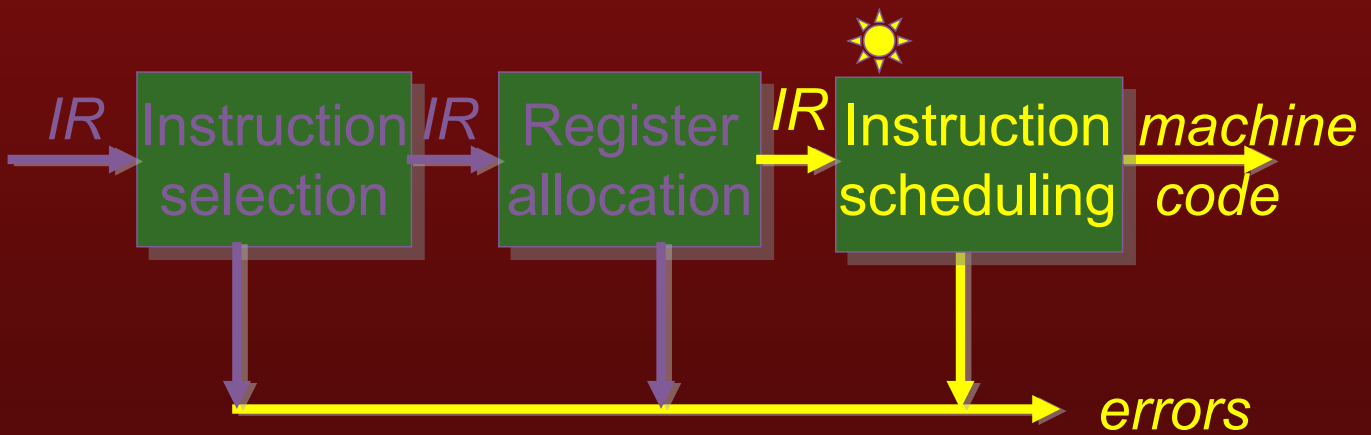


Instruction Scheduling:

- Avoid hardware stalls and interlocks.

19

The Back End

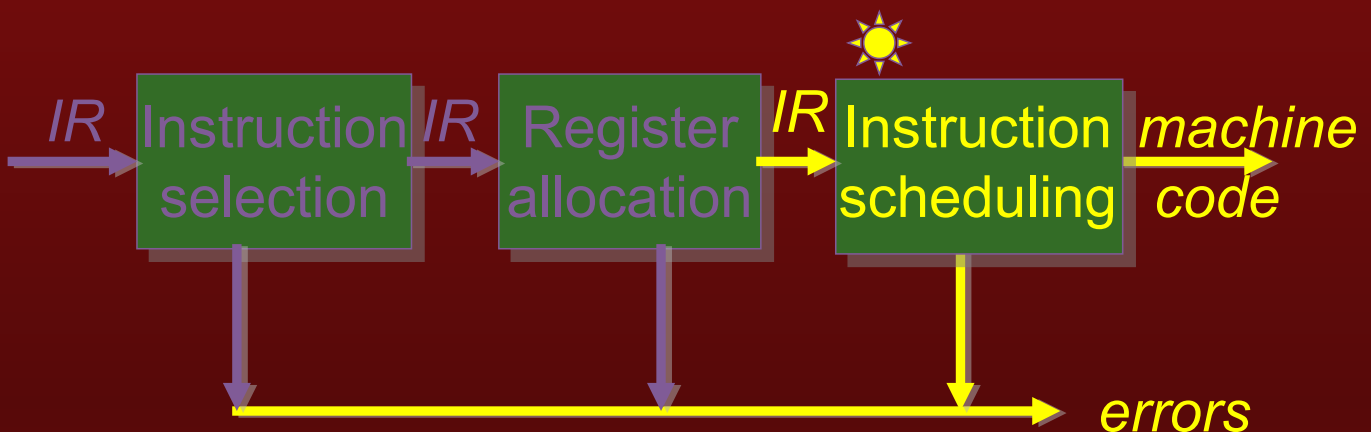


Instruction Scheduling:

- Use all functional units productively.

20

The Back End



Instruction Scheduling:

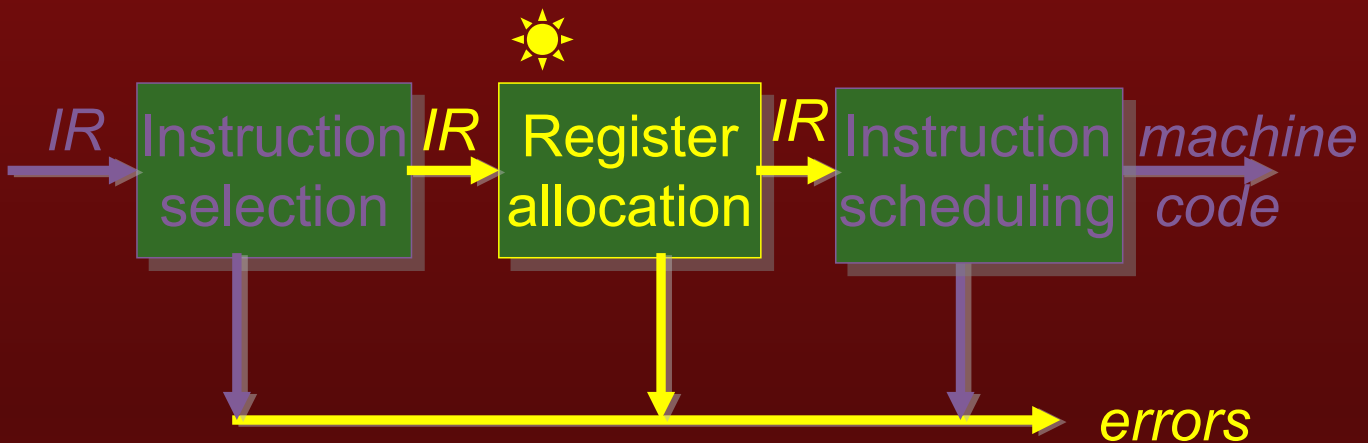
- Optimal scheduling is *NP-Complete* in nearly all cases.

21

Compiler Construction

CS 606
Sohail Aslam
Lecture 4

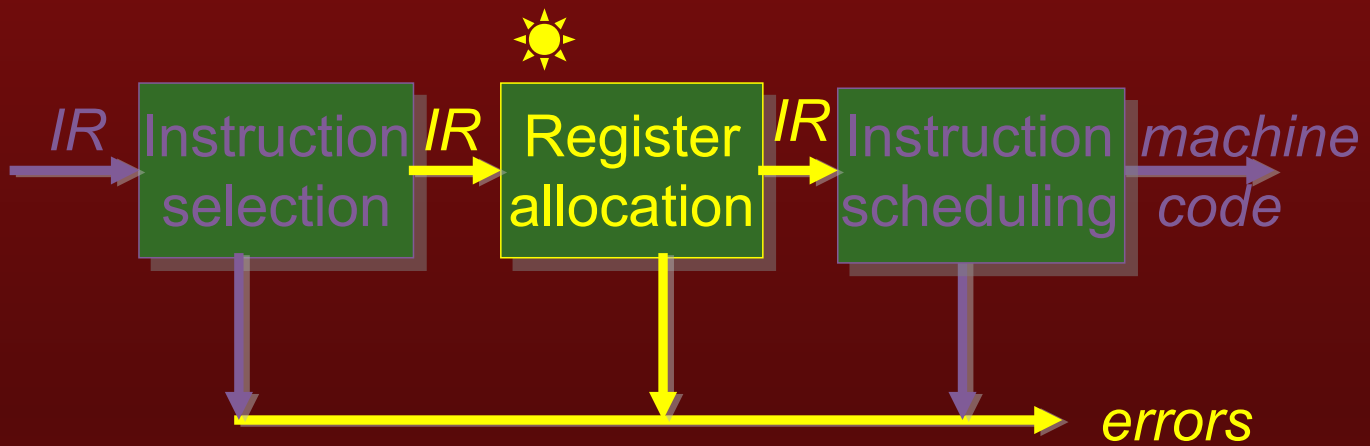
The Back End



Register Allocation:

- Have each value in a register when it is used.

The Back End

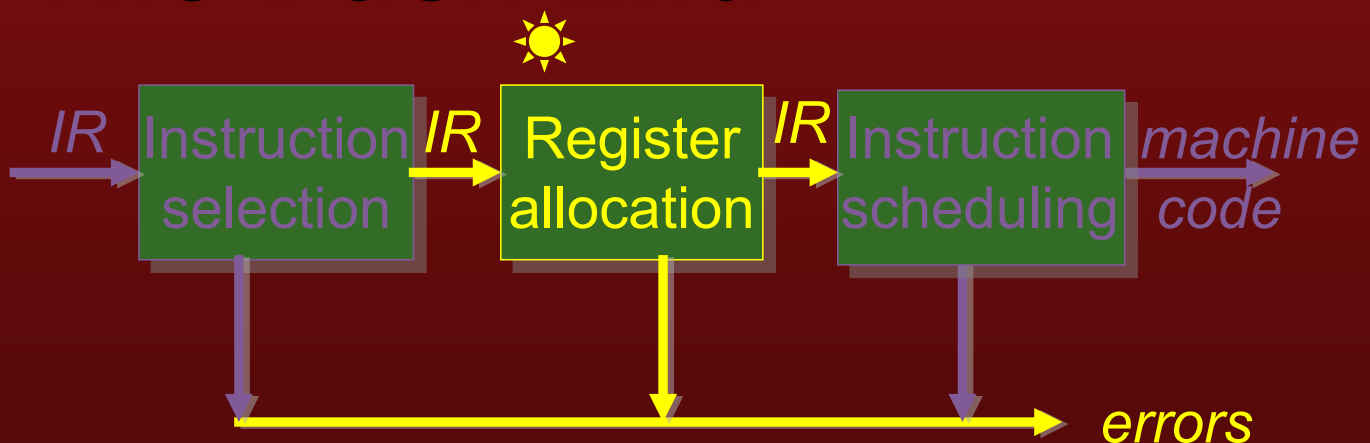


Register Allocation:

- Manage a limited set of resources – register file.

3

The Back End

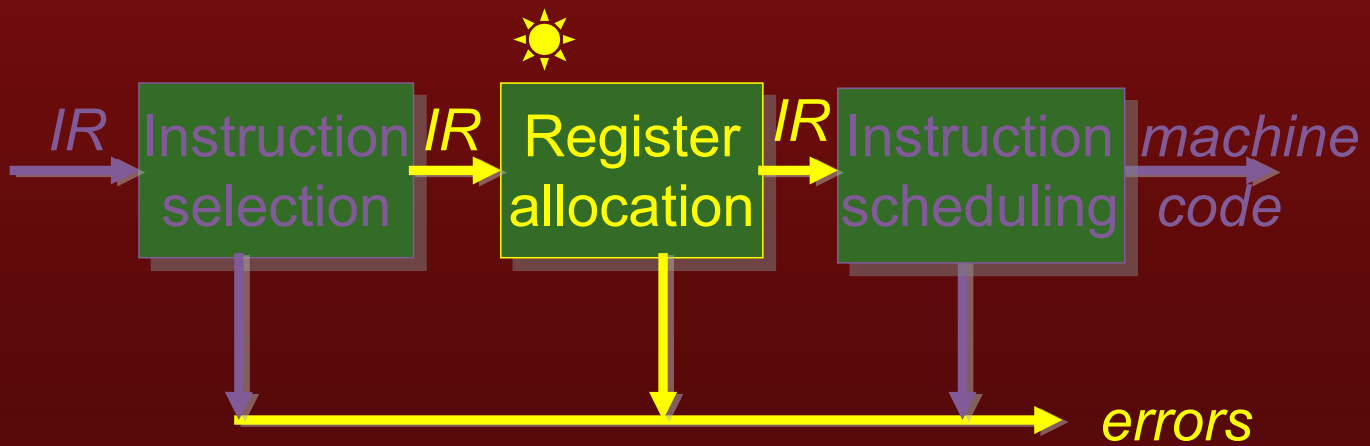


Register Allocation:

- Can change instruction choices and insert LOADs and STOREs.

4

The Back End

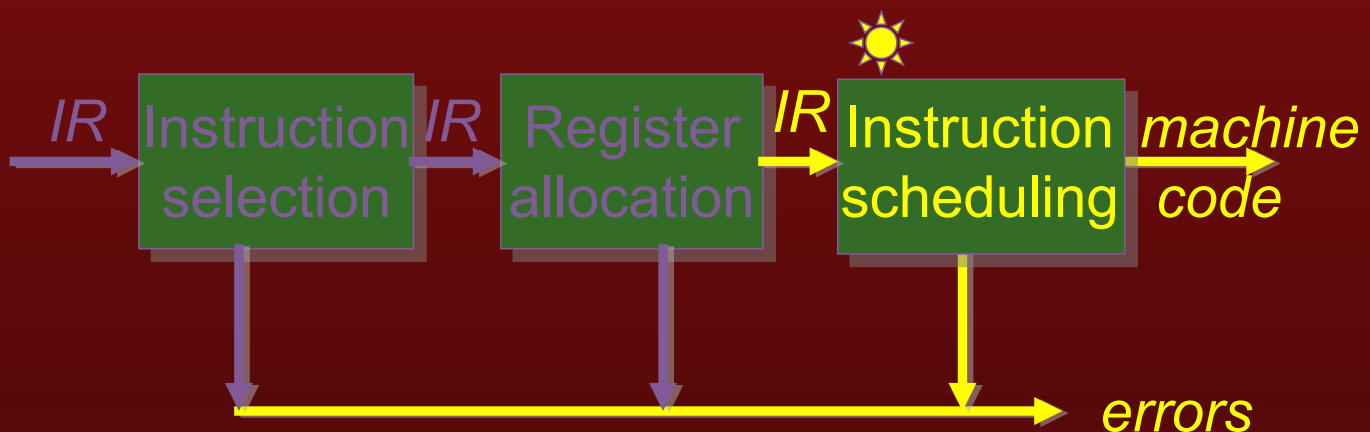


Register Allocation:

- Optimal register allocation is *NP-Complete*.

5

The Back End

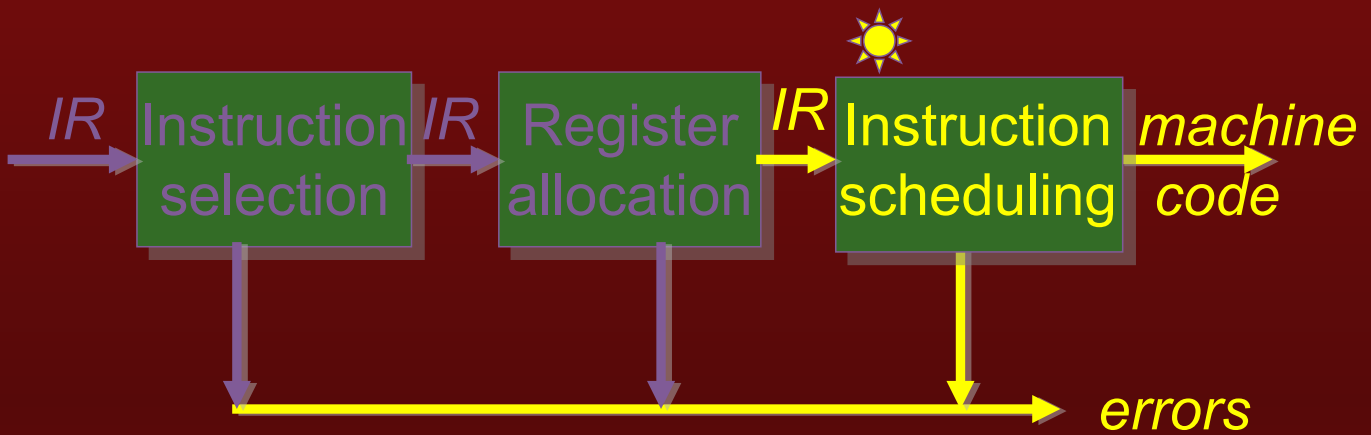


Instruction Scheduling:

- Avoid hardware stalls and interlocks.

6

The Back End

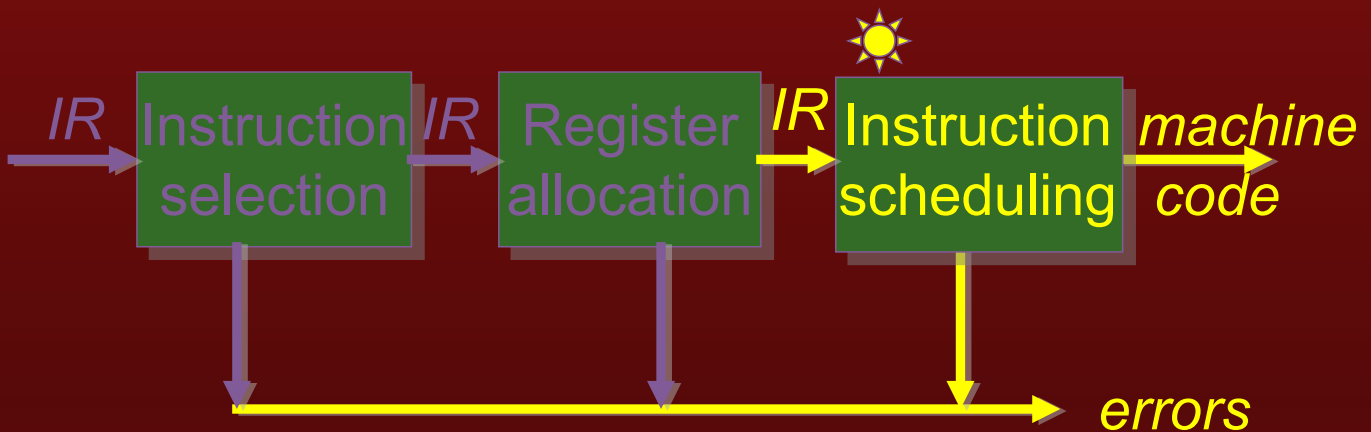


Instruction Scheduling:

- Use all functional units productively.

7

The Back End

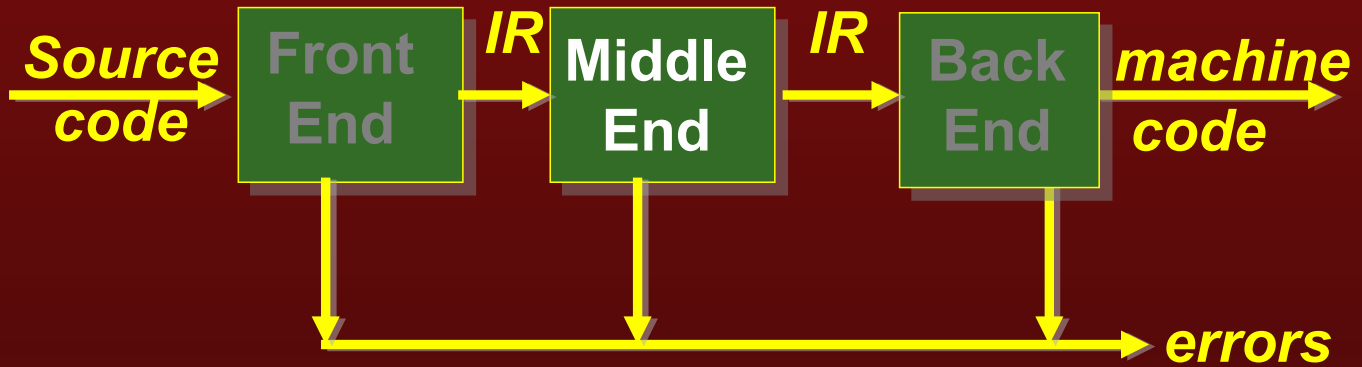


Instruction Scheduling:

- Optimal scheduling is *NP-Complete* in nearly all cases.

8

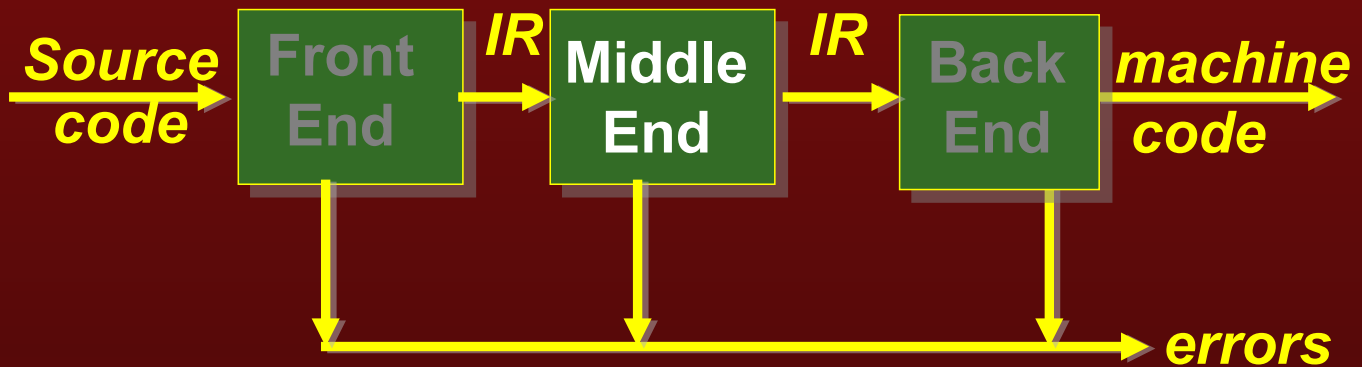
Three-pass Compiler



- Intermediate stage for code improvement or optimization

9

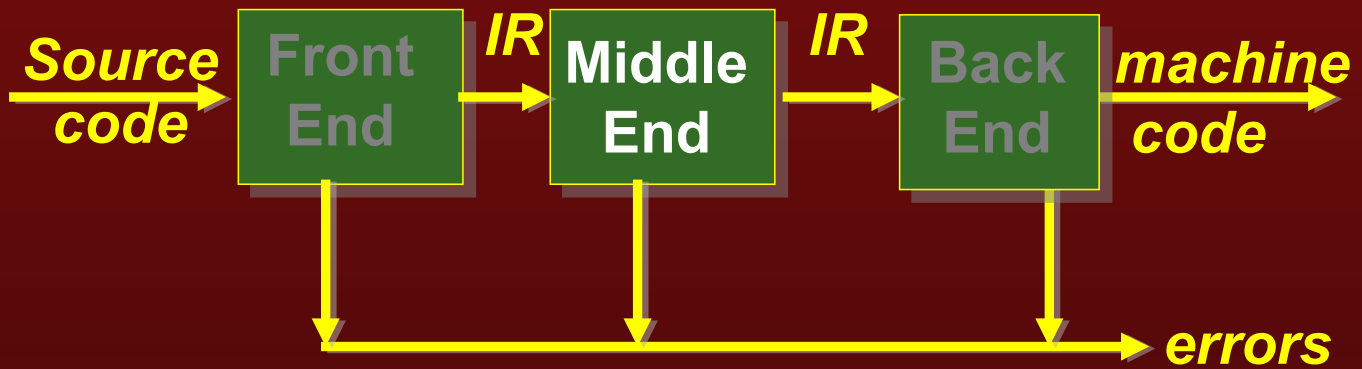
Three-pass Compiler



- Analyzes IR and rewrites (or *transforms*) IR

10

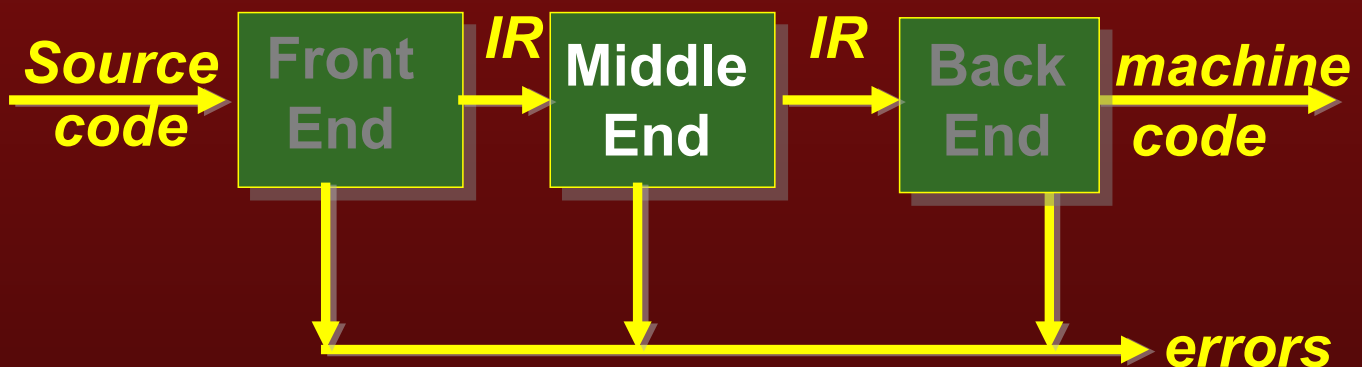
Three-pass Compiler



- Primary goal is to reduce running time of the compiled code

11

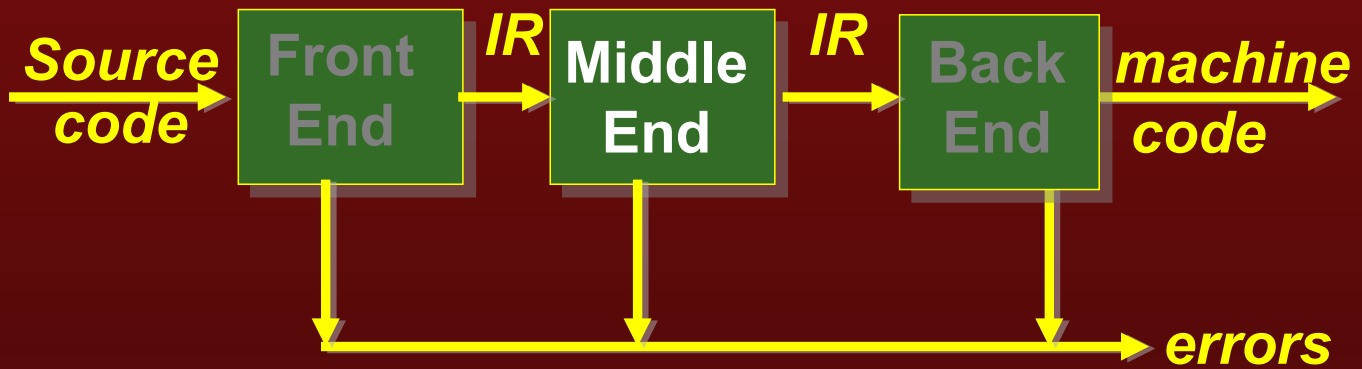
Three-pass Compiler



- May also improve space usage, power consumption, ...

12

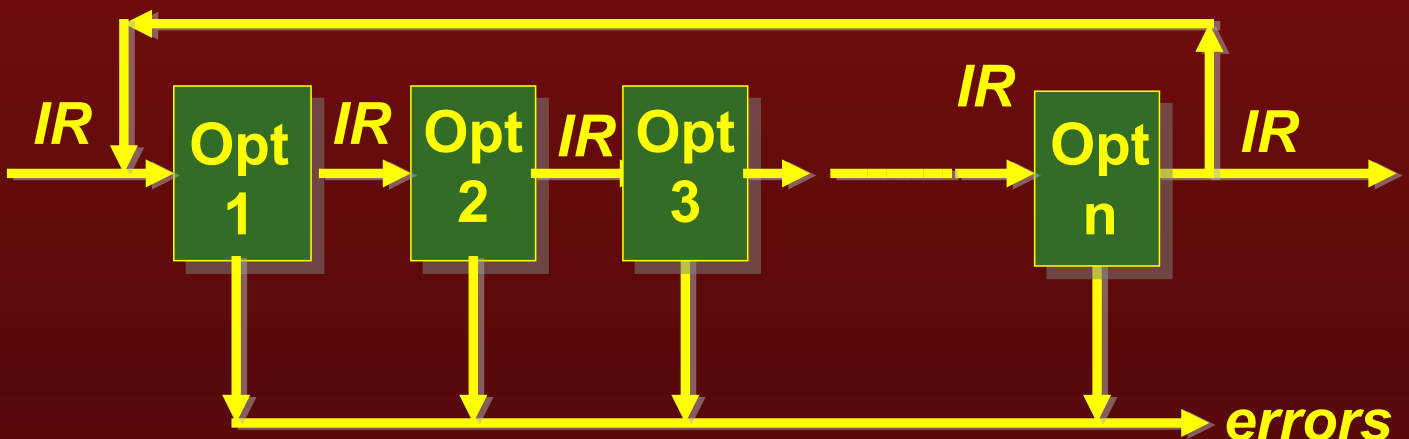
Three-pass Compiler



- Must preserve “meaning” of the code.
- Measured by values of named variables

13

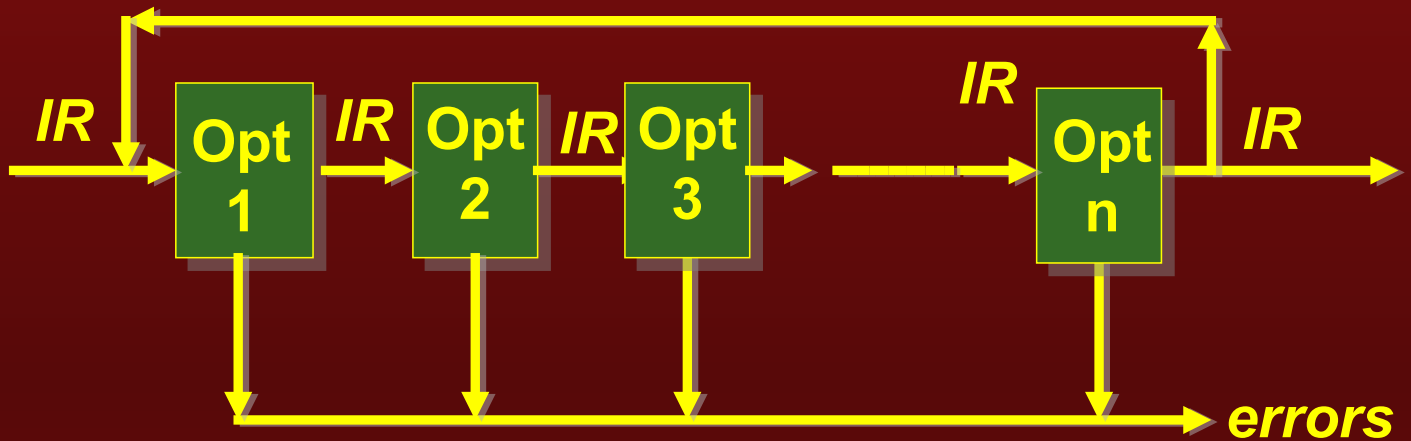
Optimizer



- Modern optimizers are structured as a **series of passes**

14

Optimizer

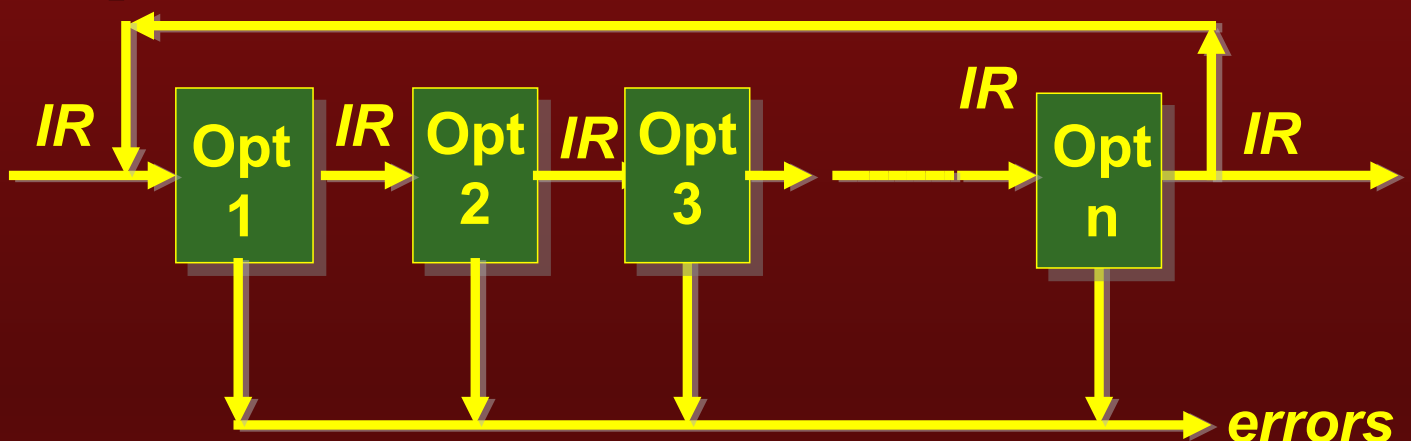


Typical transformations

- Discover & propagate some constant value

15

Optimizer

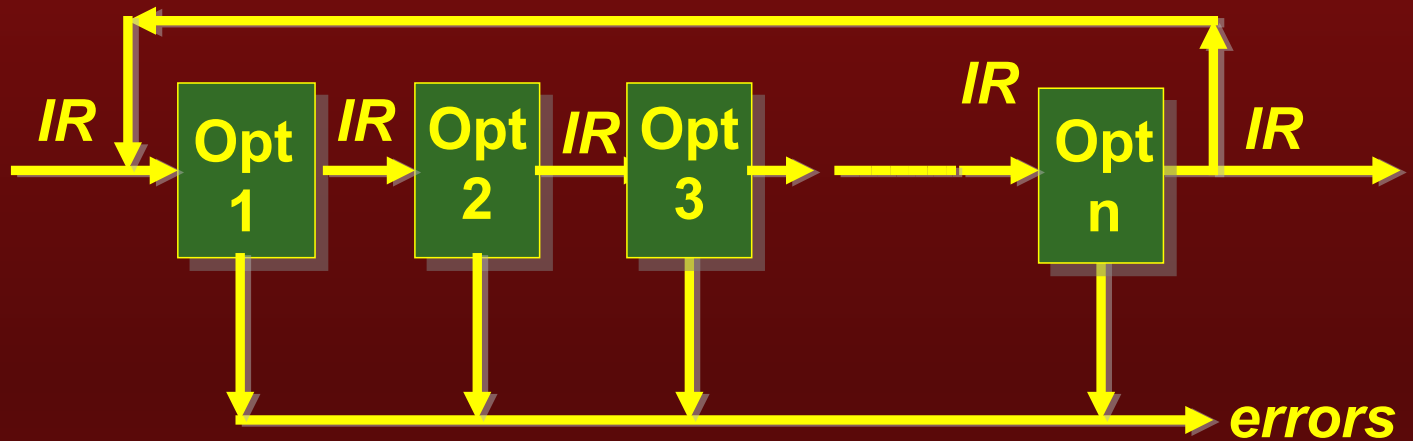


Typical transformations

- Move a computation to a less frequently executed place

16

Optimizer

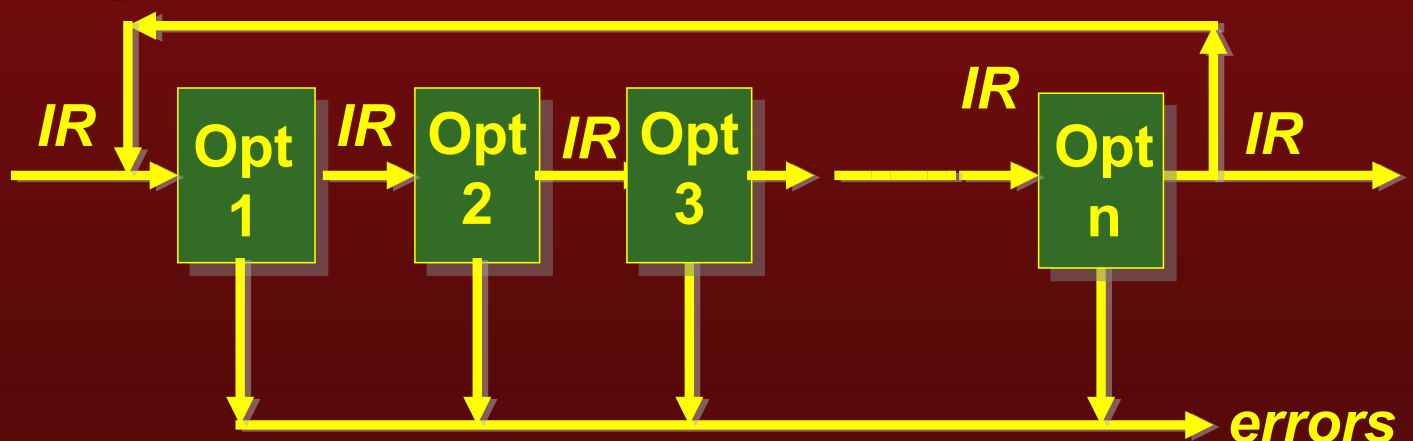


Typical transformations

- Specialize some computation based on context

17

Optimizer

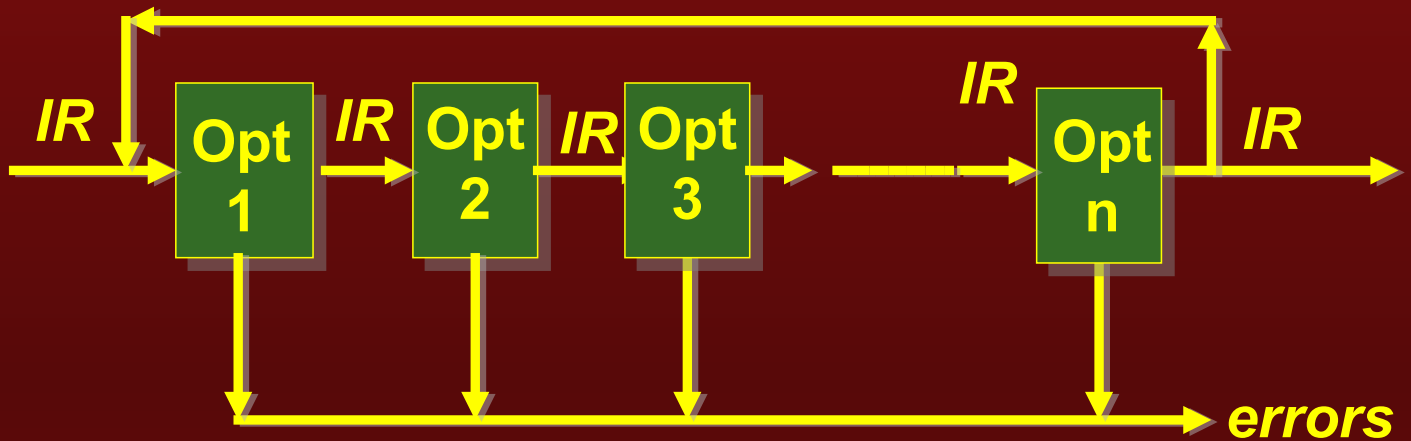


Typical transformations

- Discover a redundant computation & remove it

18

Optimizer

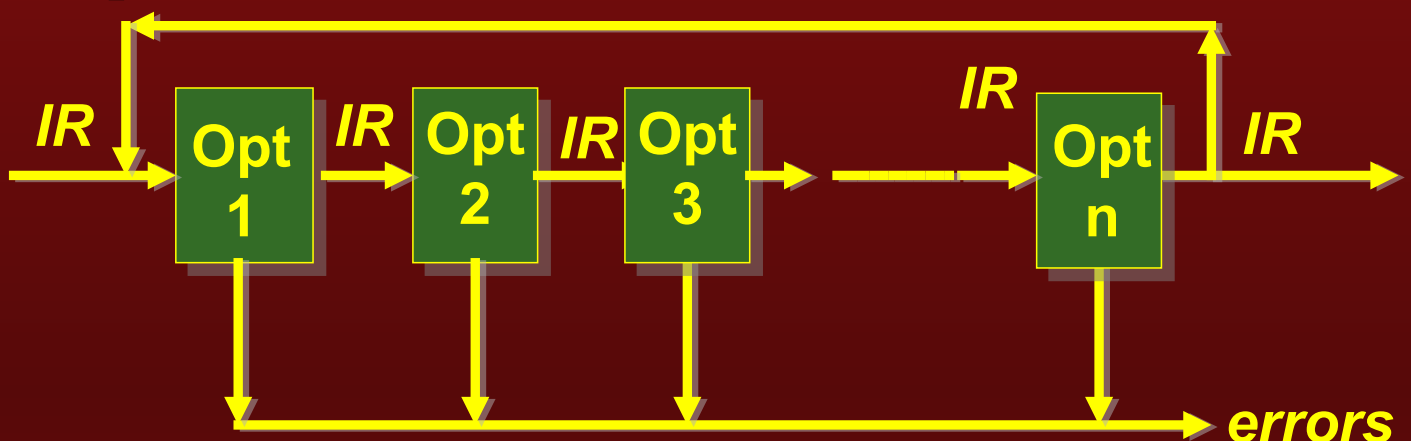


Typical transformations

- Remove useless or unreachable code

19

Optimizer



Typical transformations

- Encode an idiom in some particularly efficient form

20

Role of Run-time System

- Memory management
- Allocate/deallocate
- Garbage collection
- Run-time type checking
- Error/exception processing
- Interface to OS – I/O
- Support for parallelism
- *Parallel threads*
- *Communication and synchronization*

21

Related to Compilers

- Interpreters (direct execution)
- Assemblers
- Preprocessors
- Text formatters (non-WYSIWYG)
- Analysis tools

22

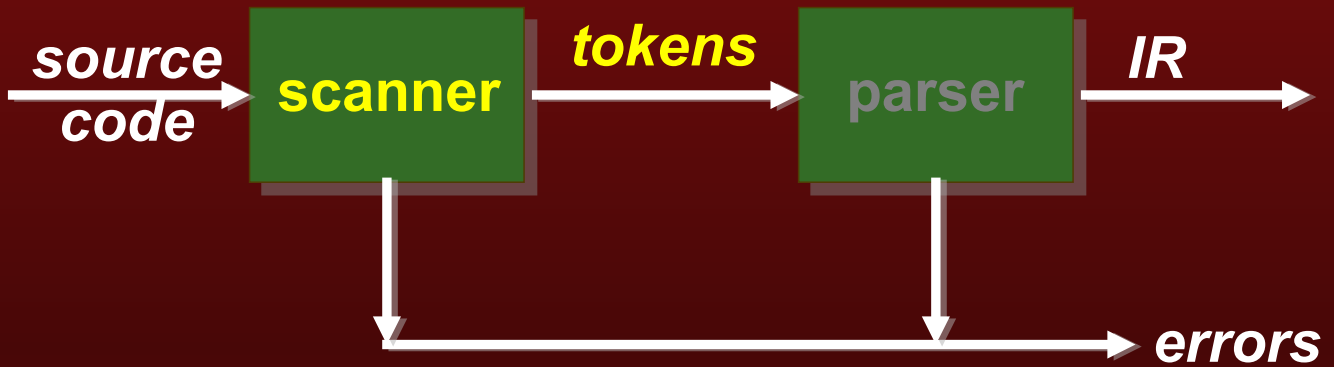
Compiler Construction

Sohail Aslam

Lecture 5

Lexical Analysis

Recall: Front-End



- Output of lexical analysis is a stream of tokens

3

Tokens

Example:

```
if( i == j )  
    z = 0;  
else  
    z = 1;
```

4

Tokens

- Input is just a sequence of characters:

i	f	(	\b	i	\b	=	=	\b	j	\n	\t	

5

Tokens

Goal:

- partition** input string into substrings
- classify** them according to their role

6

Tokens

- A token is a **syntactic category**
- Natural language:
“He wrote the program”
- Words: “He”, “wrote”, “the”, “program”

7

Tokens

- Programming language:
“if(b == 0) a = b”
- Words:
“if”, “(”, “b”, “==”, “0”,
“)”, “a”, “=”, “b”

8

Tokens

- **Identifiers:** x y11 maxsize
- **Keywords:** if else while for
- **Integers:** 2 1000 -44 5L
- **Floats:** 2.0 0.0034 1e5
- **Symbols:** () + * / { } < > ==
- **Strings:** “enter x” “error”

9

Ad-hoc Lexer

- **Hand-write** code to generate tokens.
- Partition the input string by reading left-to-right, recognizing one token at a time

10

Ad-hoc Lexer

- **Look-ahead** required to decide where one token ends and the next token begins.

11

Ad-hoc Lexer

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }
}
```

12

Ad-hoc Lexer

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }
}
```

13

Ad-hoc Lexer

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }
}
```

14

Ad-hoc Lexer

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }
}
```

15

Ad-hoc Lexer

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }
}
```

16

Ad-hoc Lexer

```
Token nextToken() {  
    if( idChar(next) )  
        return readId();  
    if( number(next) )  
        return readNumber();  
    if( next == '"' )  
        return readString();  
    ...  
    ...  
}
```

17

Ad-hoc Lexer

```
Token nextToken() {  
    if( idChar(next) )  
        return readId();  
    if( number(next) )  
        return readNumber();  
    if( next == '"' )  
        return readString();  
    ...  
    ...  
}
```

18

Ad-hoc Lexer

```
Token nextToken() {  
    if( idChar(next) )  
        return readId();  
    if( number(next) )  
        return readNumber();  
    if( next == '"' )  
        return readString();  
    ...  
    ...  
}
```

19

Ad-hoc Lexer

```
Token nextToken() {  
    if( idChar(next) )  
        return readId();  
    if( number(next) )  
        return readNumber();  
    if( next == '"' )  
        return readString();  
    ...  
    ...  
}
```

20

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

21

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

22

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

23

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

24

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

25

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

26

Ad-hoc Lexer

```
Token readId() {  
    string id = "";  
    while(true){  
        char c = input.read();  
        if(idChar(c) == false)  
            return  
                new Token(TID,id);  
        id = id + string(c);  
    }  
}
```

27

Ad-hoc Lexer

```
boolean idChar(char c)  
{  
    if( isAlpha(c) )  
        return true;  
    if( isDigit(c) )  
        return true;  
    if( c == '_' )  
        return true;  
  
    return false;  
}
```

28

Ad-hoc Lexer

```
Token readNumber() {  
    string num = "";  
    while(true) {  
        next = input.read();  
        if( !isNumber(next) )  
            return  
                new Token(TNUM,num) ;  
        num = num+string(next) ;  
    }  
}
```

29

Ad-hoc Lexer

```
Token readNumber() {  
    string num = "";  
    while(true) {  
        next = input.read();  
        if( !isNumber(next) )  
            return  
                new Token(TNUM,num) ;  
        num = num+string(next) ;  
    }  
}
```

30

Ad-hoc Lexer

```
Token readNumber() {  
    string num = "";  
    while (true) {  
        next = input.read();  
        if ( !isNumber(next) )  
            return  
                new Token (TNUM, num) ;  
        num = num + string(next) ;  
    }  
}
```

31

Ad-hoc Lexer

Problems:

- Do not know what **kind** of token we are going to read from seeing **first character**.

32

Ad-hoc Lexer

Problems:

- If token begins with “i”, is it an identifier “i” or keyword “if”?
- If token begins with “=”, is it “=” or “==”?

33

Ad-hoc Lexer

- Need a more principled approach
- Use **lexer generator** that generates efficient tokenizer automatically.

34

Compiler Construction

Sohail Aslam

Lecture 6

How to Describe Tokens?

- *Regular Languages* are the most popular for specifying tokens
 - Simple and useful theory
 - Easy to understand
 - Efficient implementations

Languages

- Let Σ be a set of characters. Σ is called the *alphabet*.
- A *language over Σ* is set of strings of characters drawn from Σ .

3

Example of Languages

Alphabet = English characters

Language = English sentences

Alphabet = ASCII

Language = C++ programs,
Java, C#

4

Notation

- Languages are **sets of strings** (finite sequence of characters)
- Need some **notation** for specifying which sets we want

5

Notation

- For lexical analysis we care about ***regular languages***.
- Regular languages can be described using ***regular expressions***.

6

Regular Languages

- Each regular expression is a notation for a regular language (a set of words).
- If **A** is a regular expression, we write **L(A)** to refer to language denoted by **A**.

7

Regular Expression

- A *regular expression* (**RE**) is defined inductively
 - a** ordinary character from Σ
 - ϵ the empty string

8

Regular Expression

$R|S$ = either R or S

RS = R followed by S
(concatenation)

R^* = concatenation of R
zero or more times
($R^* = \varepsilon | R|RR|RRR\dots$)

9

RE Extensions

$R?$ = $\varepsilon | R$ (zero or one R)

R^+ = RR^* (one or more R)

(R) = R (grouping)

10

RE Extensions

$[abc]$ = $a|b|c$ (any of listed)

$[a-z]$ = $a|b|...|z$ (range)

$[^ab]$ = $c|d|...$ (anything but
‘a’‘b’)

11

Regular Expression

RE	Strings in $L(R)$
a	“a”
ab	“ab”
$a b$	“a” “b”
$(ab)^*$	“” “ab” “abab” ...
$(a \epsilon)b$	“ab” “b”

12

Example: integers

- **integer**: a non-empty string of digits
- **digit** = '0'|'1'|'2'|'3'|'4'|
'5'|'6'|'7'|'8'|'9'
- **integer** = digit digit*

13

Example: identifiers

- **identifier**:
string of letters or digits
starting with a letter
- **C identifier**:
[a-zA-Z_][a-zA-Z0-9_]*

14

Recap

Tokens:

strings of characters representing lexical units of programs such as identifiers, numbers, operators.

15

Recap

Regular Expressions:

concise description of tokens. A regular expression describes a set of strings.

16

Recap

Language $L(R)$:

set of strings represented by a regular expression R .

$L(R)$ is the language denoted by regular expression R .

17

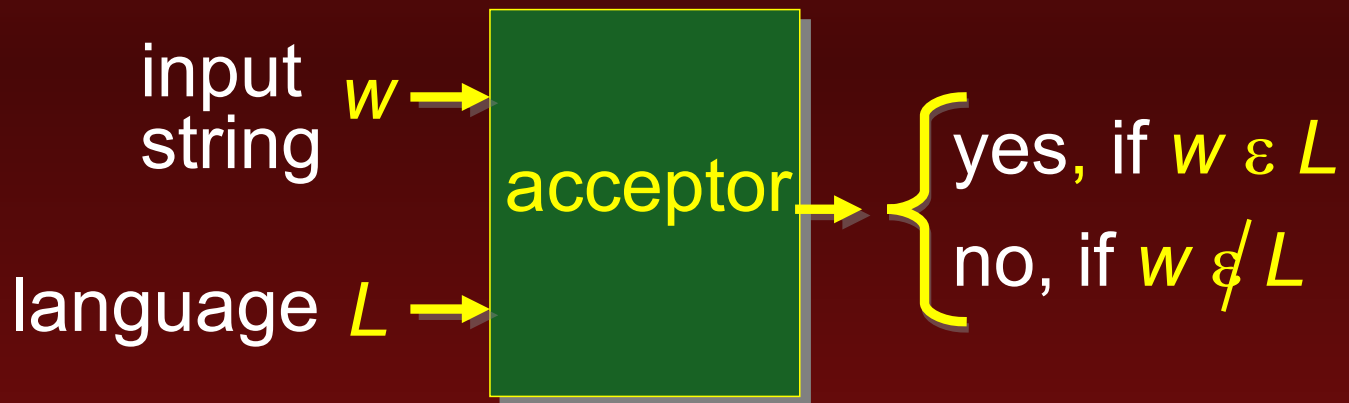
How to Use REs

- We need mechanism to determine if an input string w belongs to $L(R)$, the language denoted by regular expression R .

18

Acceptor

- Such a mechanism is called an **acceptor**.



19

Finite Automata (FA)

- Specification:**
Regular Expressions
- Implementation:**
Finite Automata

20

Finite Automata

Finite Automaton consists of

- An input alphabet (Σ)
- A set of states
- A start (initial) state
- A set of transitions
- A set of accepting (final) states

21

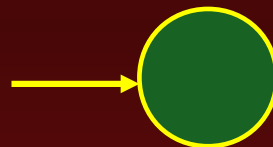
Finite Automaton

State Graphs

A state



The start state



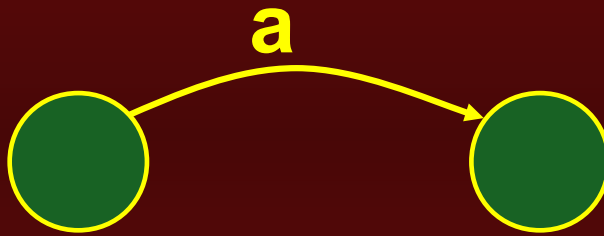
An accepting state



22

Finite Automaton

State Graphs



A transition

23

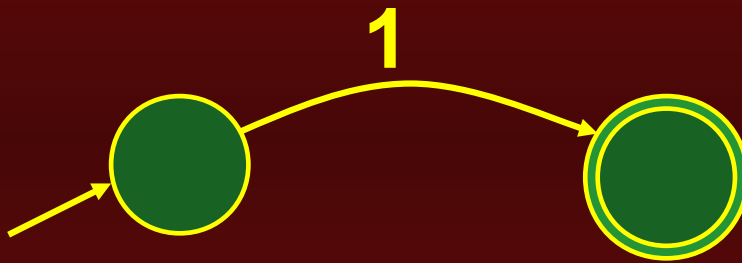
Finite Automata

- A finite automaton *accepts* a string if we can follow transitions labelled with characters in the string from start state to some accepting state.

24

FA Example

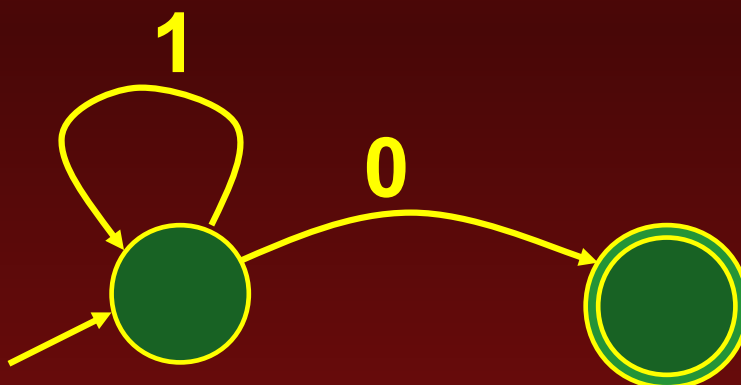
A FA that accepts only “1”



25

FA Example

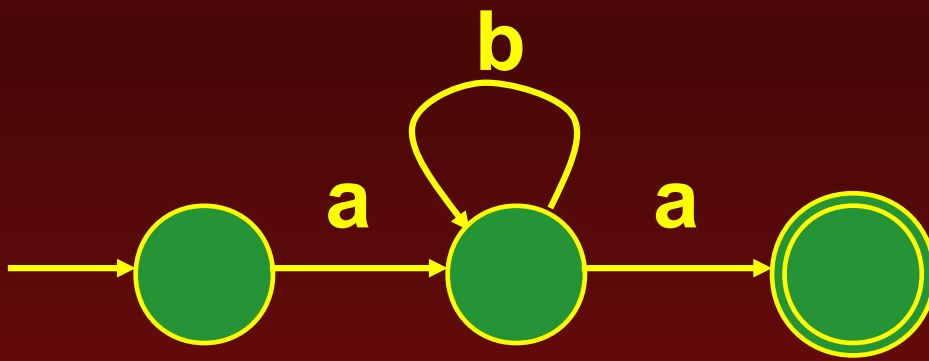
- A FA that accepts any number of 1's followed by a single 0



26

FA Example

- A FA that accepts ab^*a
- Alphabet: $\{a,b\}$



27

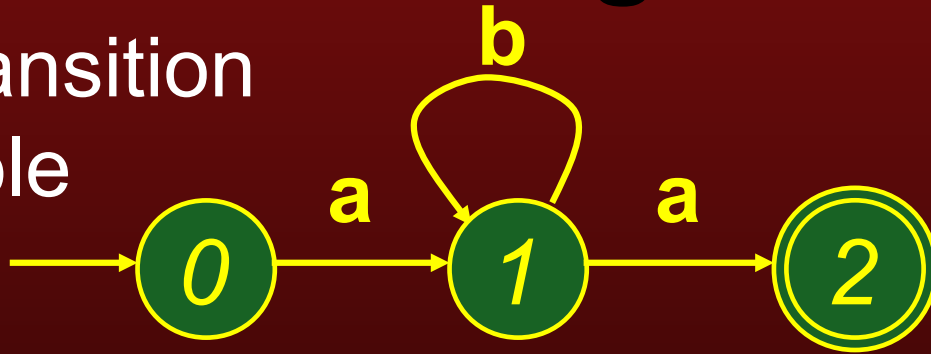
Compiler Construction

Sohail Aslam

Lecture 7

Table Encoding of FA

- Transition table



	a	b
0	1	err
1	2	1
2	err	err

2

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

3

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF ) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

4

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF ) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

5

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF ) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

6

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF ) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

7

Simulating FA

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
    c = input.read();
    if(c == EOF) break;
    state=trans_table[state][c];
}
return accept_states[state];
```

8

RE $\rightarrow$ Finite Automata

- Can we build a finite automaton for every regular expression?
- **Yes**, – build FA **inductively** based on the definition of Regular Expression

9

NFA

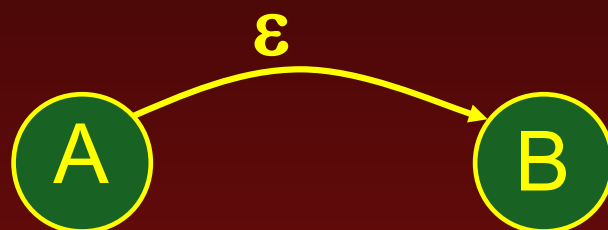
Nondeterministic Finite Automaton (NFA)

- Can have **multiple** transitions for one input in a given state
- Can have ϵ - moves

10

Epsilon Moves

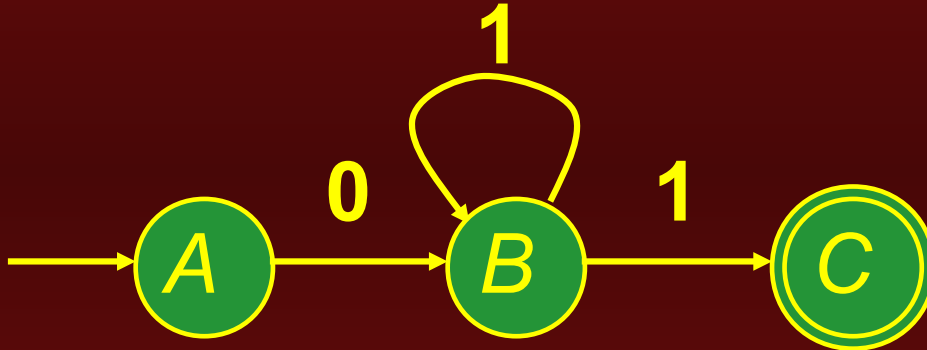
- ϵ – moves
machine can move from state **A** to state **B** without consuming input



11

NFA

operation of the automaton is not completely defined by input



On input “11”, automaton could be in either state

12

Execution of FA

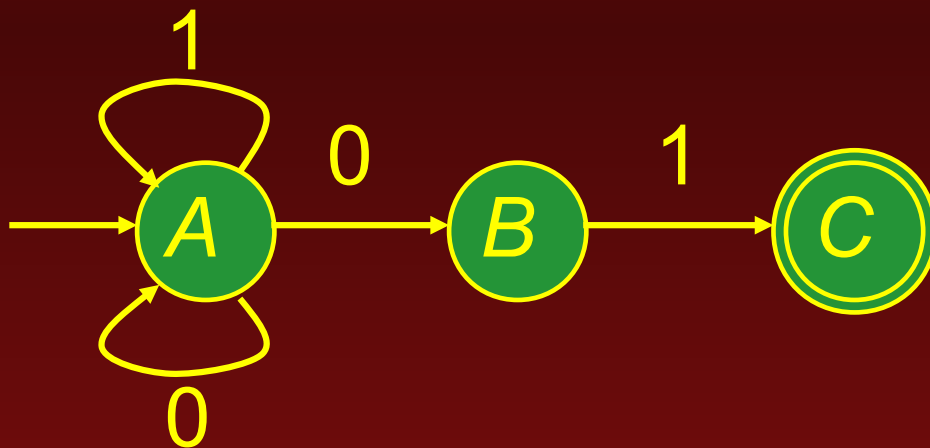
A NFA can choose

- Whether to make ϵ -moves.
- Which of multiple transitions to take for a single input.

13

Acceptance of NFA

- NFA can get into **multiple** states
- **Rule:** NFA accepts if it can get in a final state



14

DFA and NFA

Deterministic Finite Automata (DFA)

- **One** transition per input per state.
- No ϵ - moves

15

Execution of FA

A DFA

- can take *only one path* through the state graph.
- Completely determined by input.

16

NFA vs DFA

- NFAs and DFAs recognize the **same** set of languages (regular languages)
- DFAs are **easier** to implement – table driven.

17

NFA vs DFA

- For a given language, the NFA can be simpler than the DFA.
- DFA can be **exponentially larger** than NFA.

18

NFA vs DFA

- NFAs are the **key** to **automating** RE $\rightarrow$ DFA construction.

19

RE $\rightarrow$ NFA Construction

Thompson's construction (CACM 1968)

- Build an **NFA** for each RE term.
- Combine NFAs with **ϵ** -moves.

20

RE $\rightarrow$ NFA Construction

Subset construction **NFA $\rightarrow$ DFA**

- Build the simulation.
- Minimize number of states in DFA (*Hopcroft's algorithm*)

21

RE $\rightarrow$ NFA Construction

Key idea:

- NFA pattern for each symbol and each operator.
- Join them with ε -moves in precedence order.

22

RE $\rightarrow$ NFA Construction



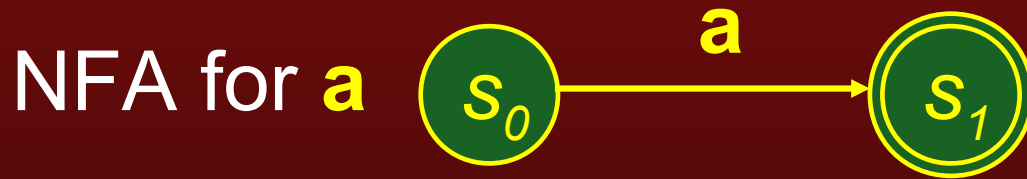
NFA for **a**



NFA for **ab**

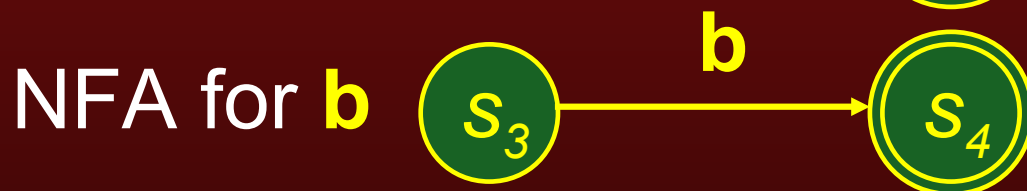
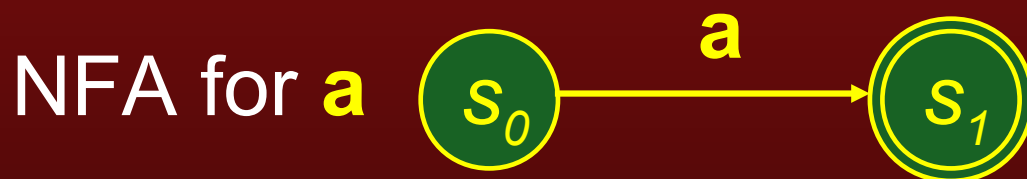
23

RE $\rightarrow$ NFA Construction



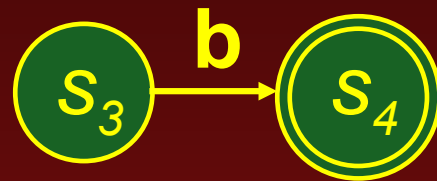
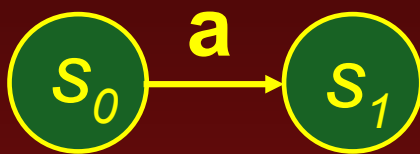
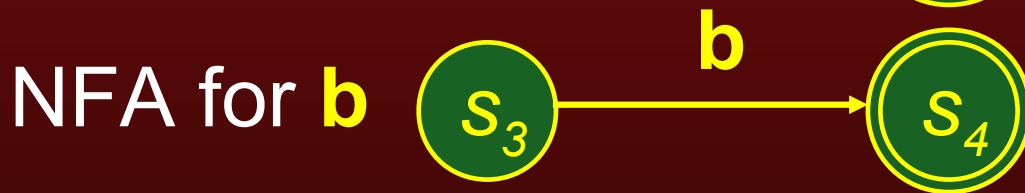
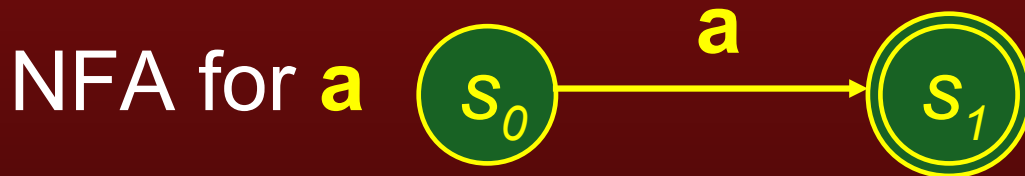
24

RE $\rightarrow$ NFA Construction



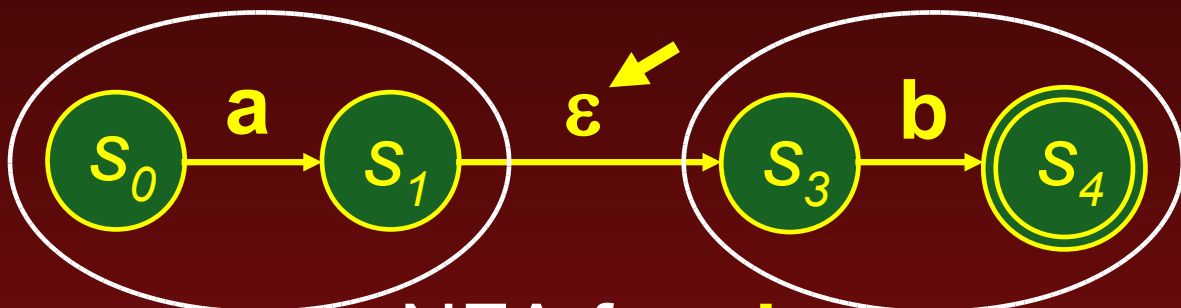
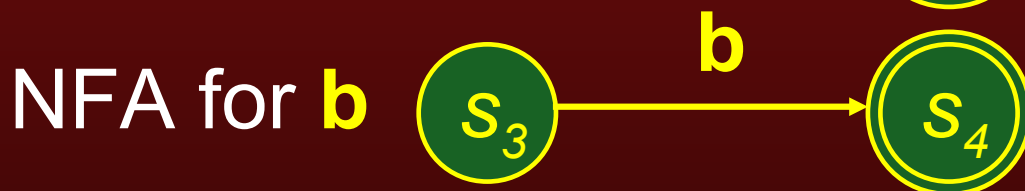
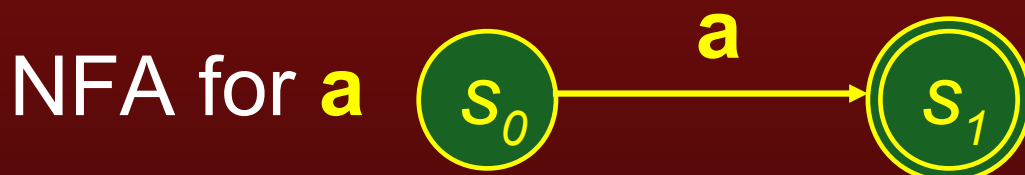
25

RE $\rightarrow$ NFA Construction



26

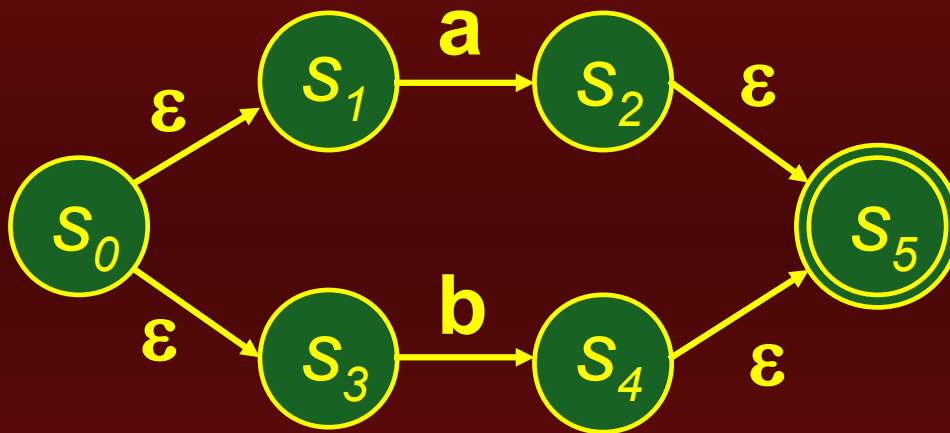
RE $\rightarrow$ NFA Construction



NFA for **ab**

27

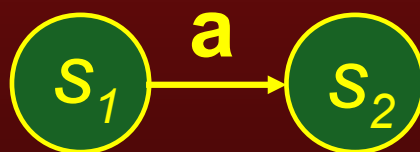
RE $\rightarrow$ NFA Construction



NFA for **a | b**

28

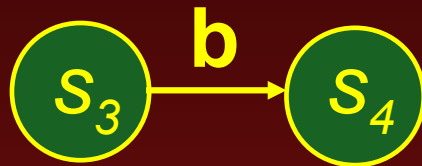
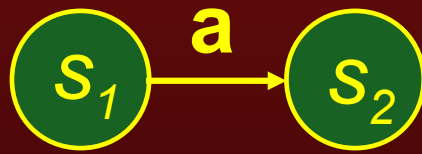
RE $\rightarrow$ NFA Construction



NFA for **a**

29

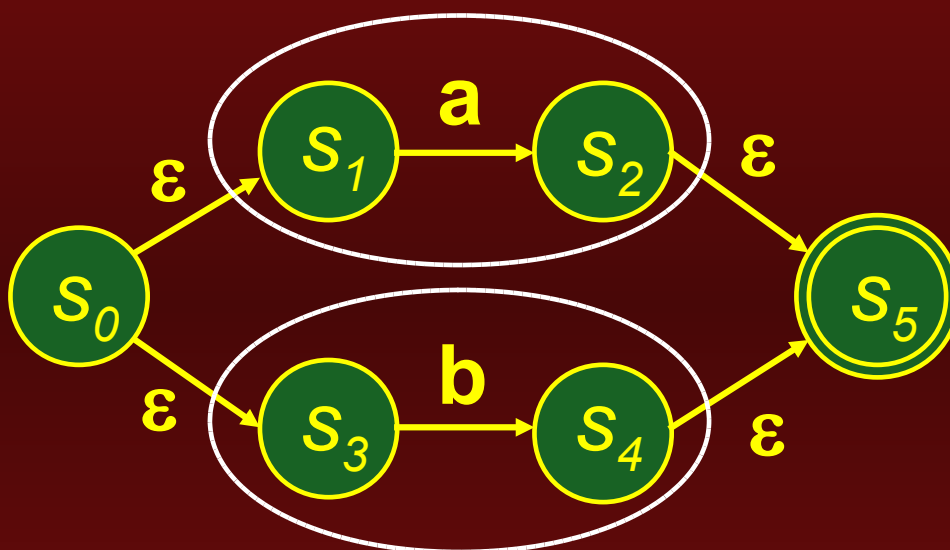
RE $\rightarrow$ NFA Construction



NFA for **a** and **b**

30

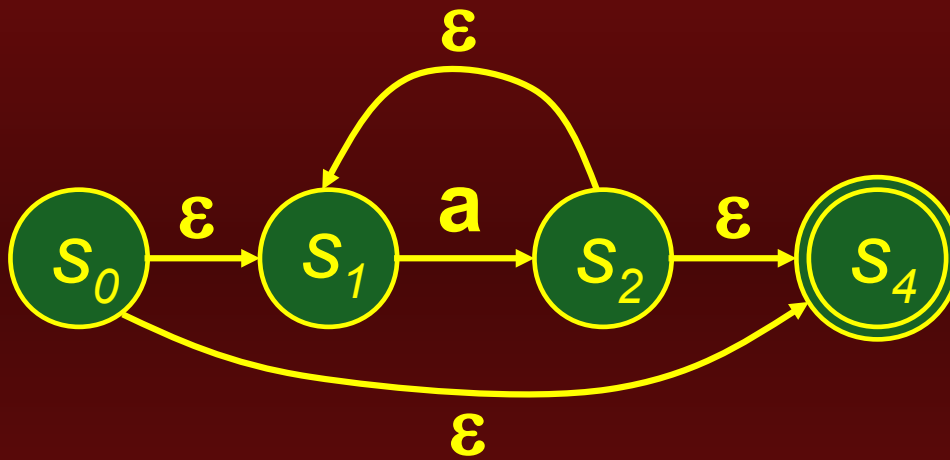
RE $\rightarrow$ NFA Construction



NFA for **a | b**

31

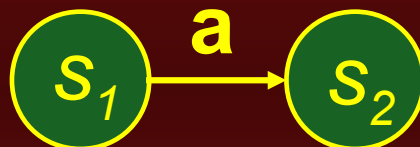
RE $\rightarrow$ NFA Construction



NFA for a^*

32

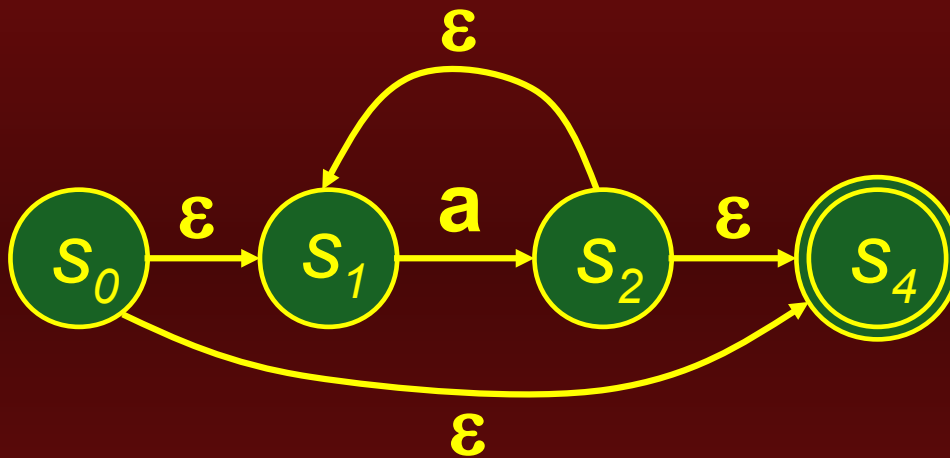
RE $\rightarrow$ NFA Construction



NFA for a

33

RE $\rightarrow$ NFA Construction

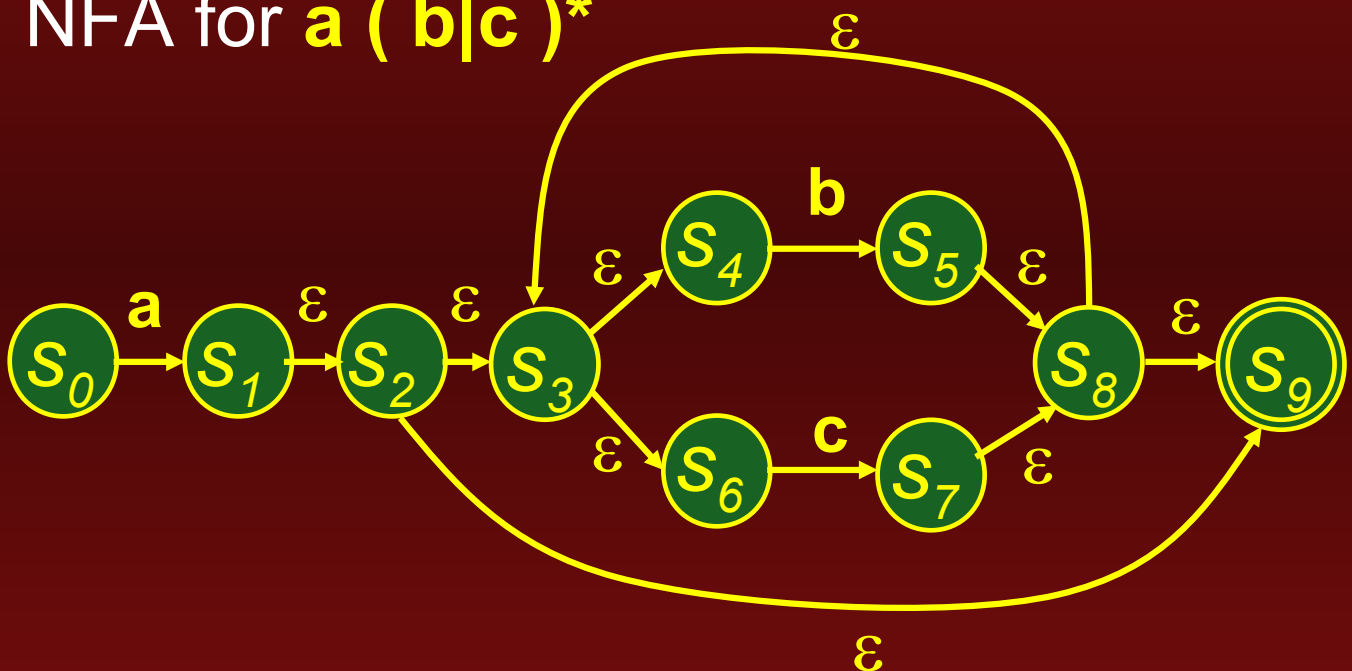


NFA for a^*

34

Example RE $\rightarrow$ NFA

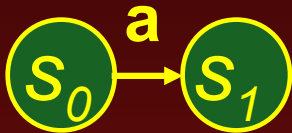
NFA for $a (b|c)^*$



35

Example RE $\rightarrow$ NFA

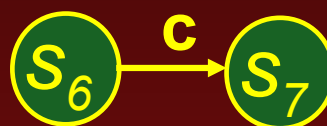
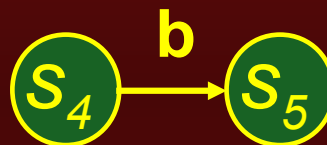
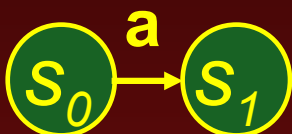
building NFA for **a (b|c)***



36

Example RE $\rightarrow$ NFA

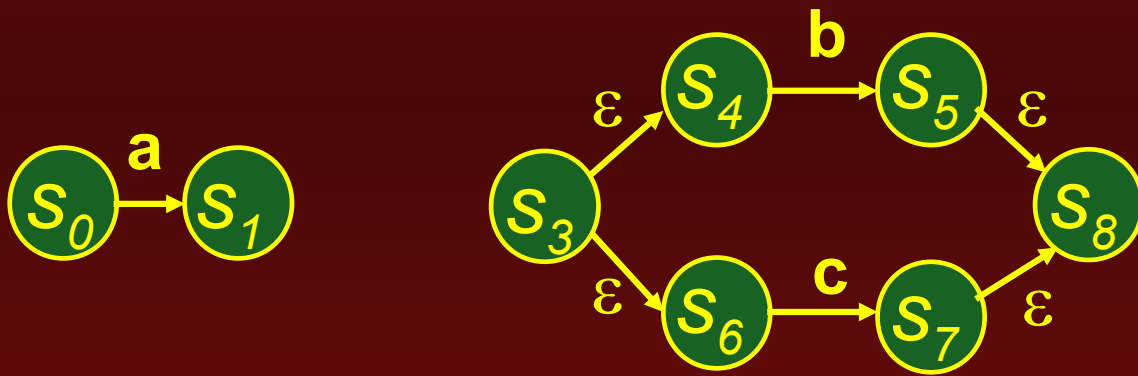
NFA for **a**, **b** and **c**



37

Example RE $\rightarrow$ NFA

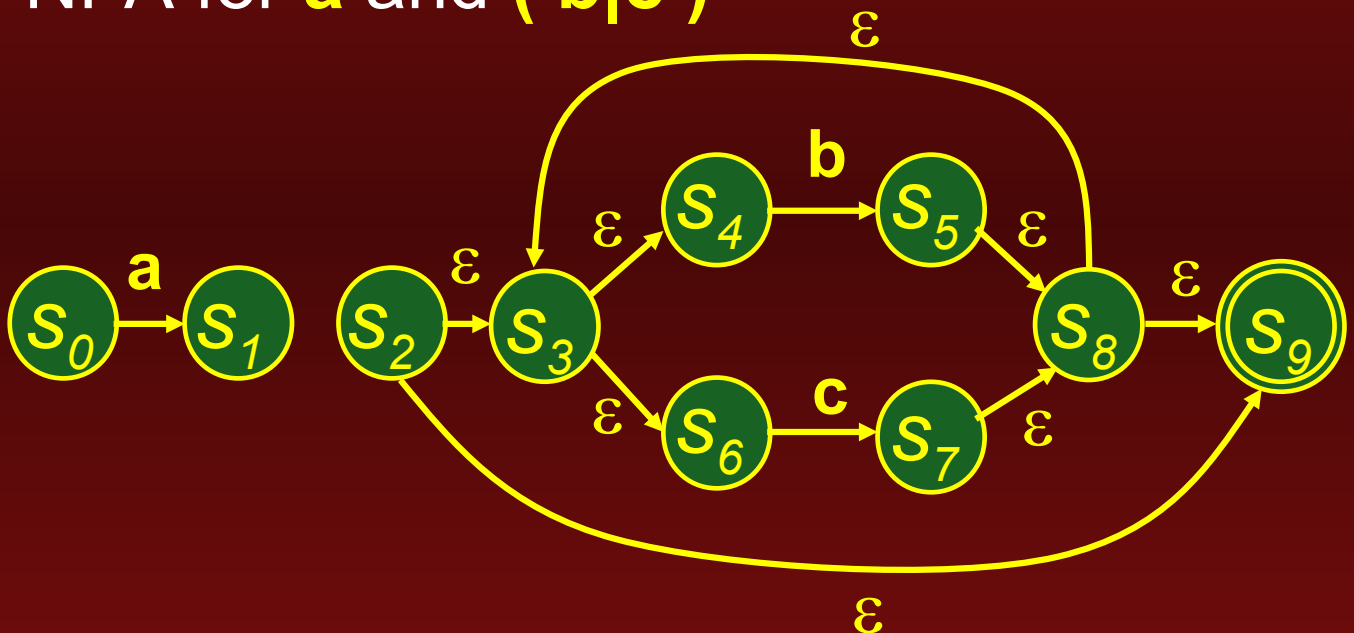
NFA for **a** and **b|c**



38

Example RE $\rightarrow$ NFA

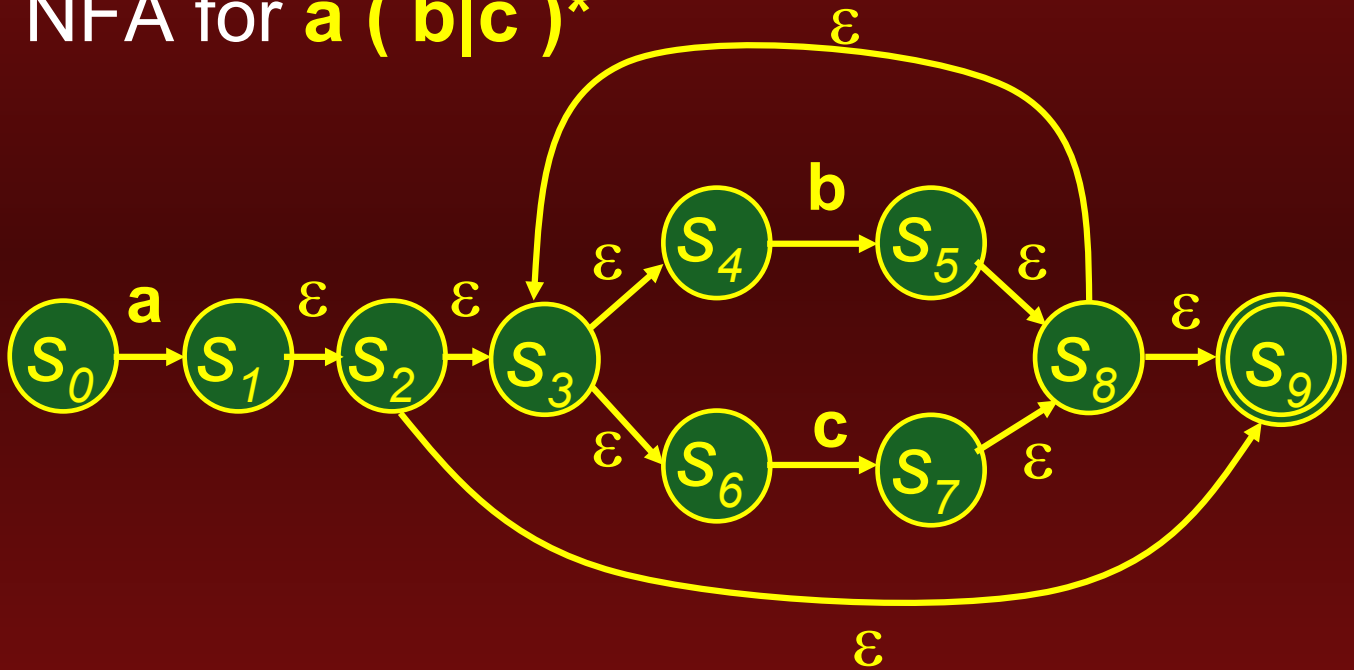
NFA for **a** and **(b|c)***



39

Example RE $\rightarrow$ NFA

NFA for $a (b|c)^*$



40

Compiler Construction

Sohail Aslam

Lecture 8

NFA $\rightarrow$ DFA Construction

- The algorithm is called *subset construction*.
- In the **transition table** of an NFA, each entry is a set of states.
- In DFA, each entry is a **single state**

2

NFA $\rightarrow$ DFA Construction

- The general idea behind NFA-to-DFA construction is that **each DFA state** corresponds to a **set of NFA states**.

3

NFA $\rightarrow$ DFA Construction

- The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.

4

NFA $\rightarrow$ DFA Construction

- We will use the following operations.
- ϵ -closure(T):
set of NFA states reachable from some NFA state s in T on ϵ -transitions alone.

5

NFA $\rightarrow$ DFA Construction

- $move(T, a)$:
set of NFA states to which there is a transition on input a from some NFA state s in set of states T .

6

NFA $\rightarrow$ DFA Construction

- Before it sees the first input symbol, NFA can be in any of the state in the set $\varepsilon\text{-closure}(s_0)$, where s_0 is the start state of the NFA.

7

NFA $\rightarrow$ DFA Construction

- Suppose that exactly the states in set T are reachable from s_0 on a given sequence of input symbols.

8

NFA $\rightarrow$ DFA Construction

- Let a be the next input symbol.
- On seeing a , the NFA can move to any of the states in the set $move(T, a)$.

9

NFA $\rightarrow$ DFA Construction

- Let a be the next input symbol.
- On seeing a , the NFA can move to any of the states in the set $move(T, a)$.

10

NFA $\rightarrow$ DFA Construction

- When we allow for ϵ -transitions, NFA can be in any of the states in $\epsilon\text{-closure}(move(T, a))$ after seeing a .

11

Subset Construction

Algorithm:

Input:

NFA N with state set S ,
alphabet Σ , start state
 s_0 , final states F

12

Subset Construction

Output:

DFA D with state set S' ,
alphabet Σ , start states
 $s_0' = \varepsilon\text{-closure}(s_0)$, final
states F' , transition
table: $S' \times \Sigma \rightarrow S'$

13

Algorithm

// initially, ϵ -closure(s_0) is the only

// state in D states S' and it is

// unmarked

$s_0' = \epsilon$ -closure(s_0)

$S' = \{s_0'\}$ (unmarked)

14

◆ *while* there is some unmarked state T
 in S'
 mark state T
 for all a in Σ *do*
 $U = \epsilon$ -closure(move(T, a));
 if U not already in S'
 add U as an unmarked state to S'
 $D\text{tran}(T, a) = U$;
 end for
end while

15

Algorithm

◆ F' :

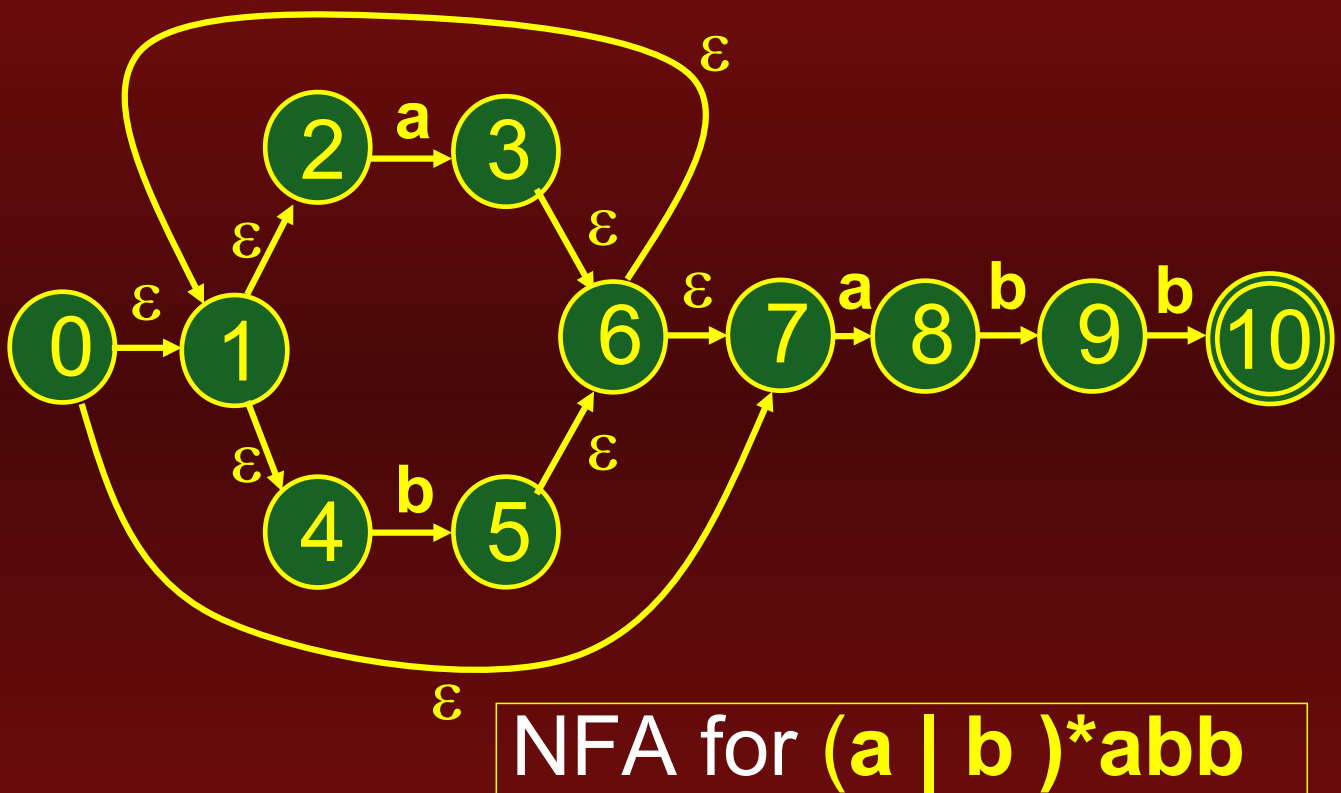
for each DFA state S

if S contains an NFA final state

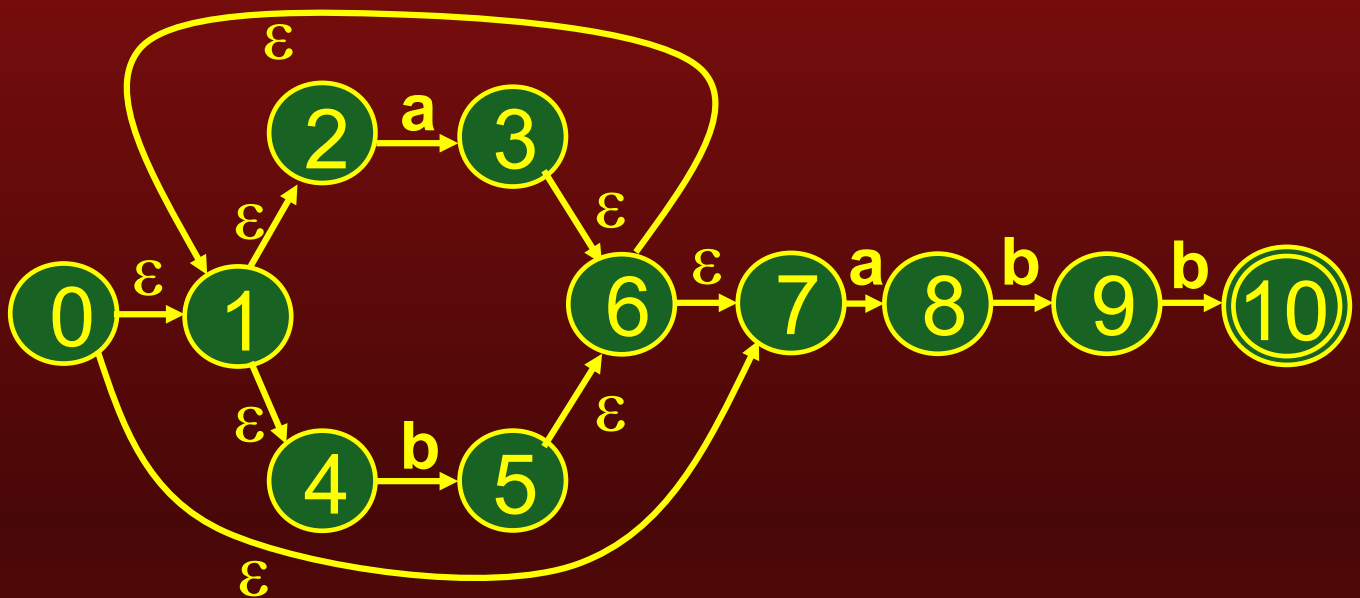
mark S as DFA final state

16

Subset Construction Example

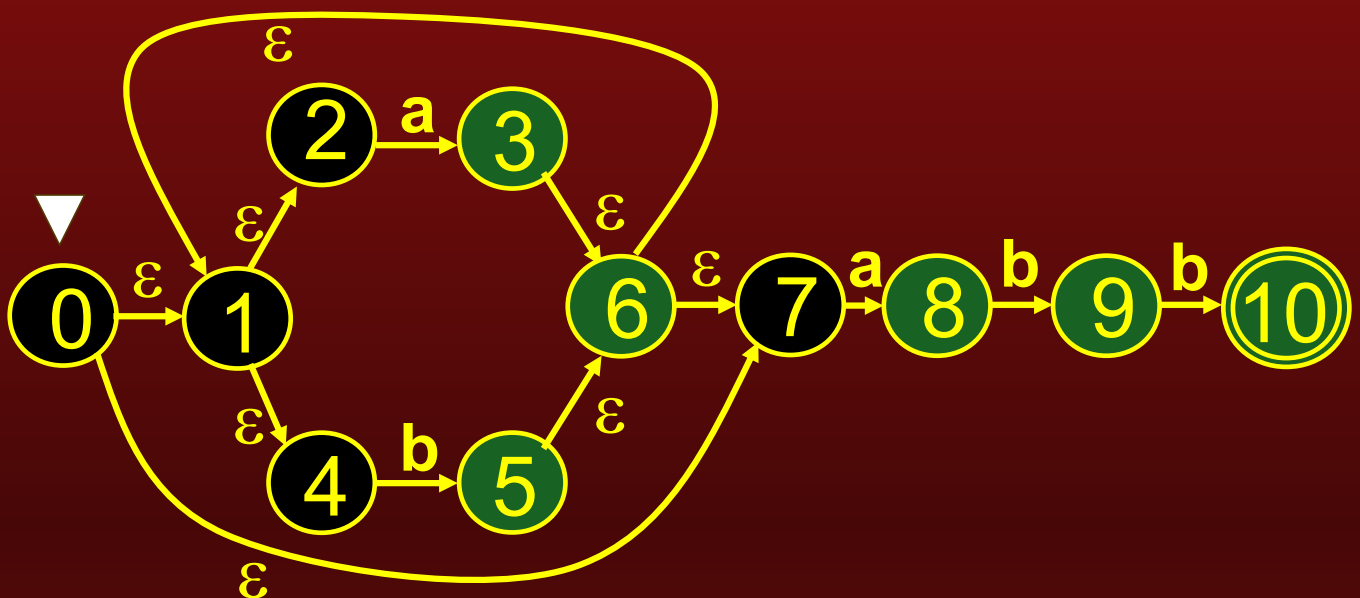


17



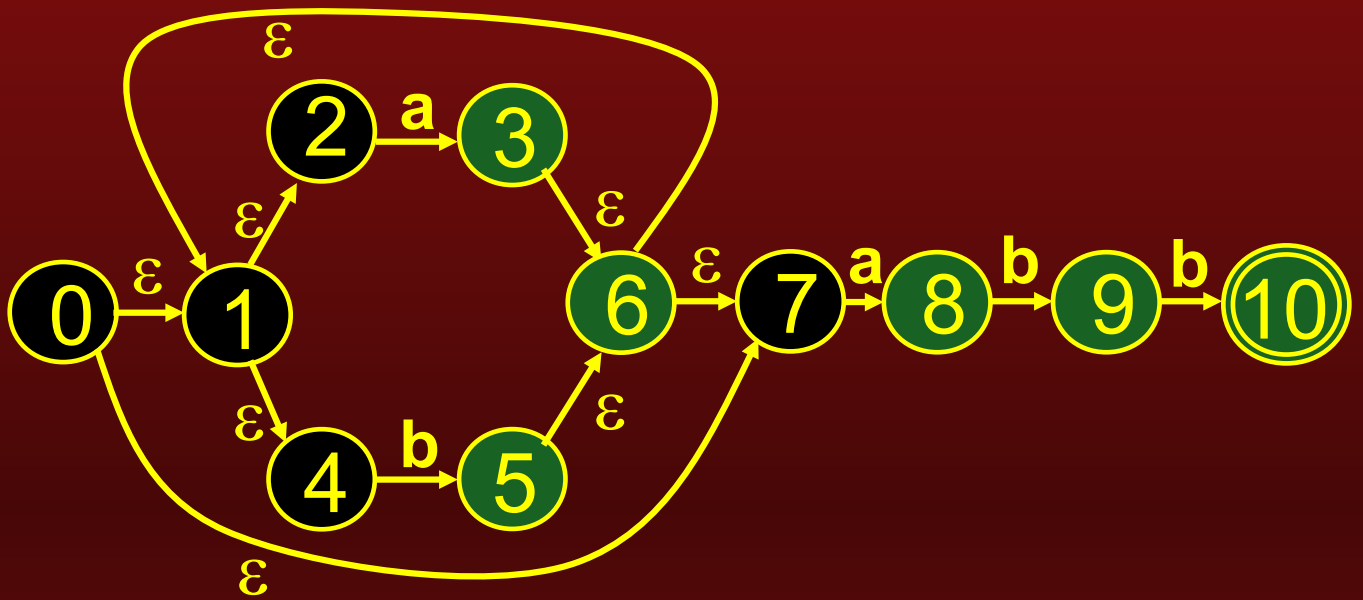
- The start state of equivalent DFA is ϵ -closure(0), which is $A = \{0, 1, 2, 4, 7\}$

18



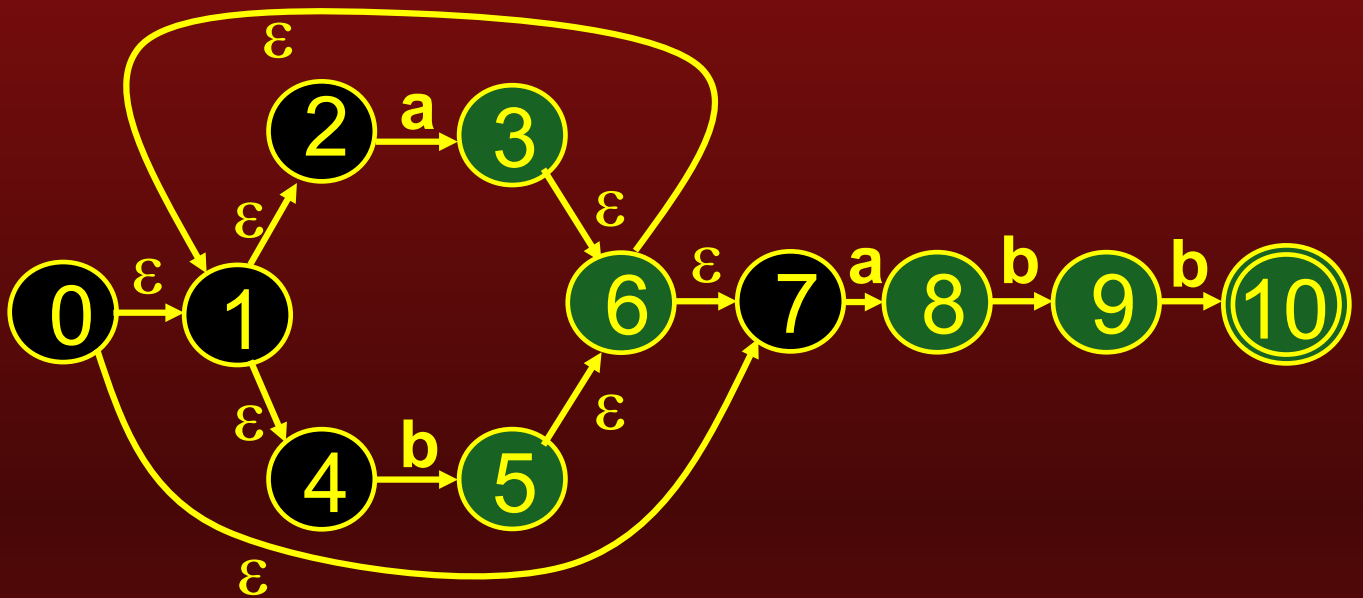
- $A = \{0, 1, 2, 4, 7\}$, these are exactly the states reachable from state 0 via ϵ -transition.

19



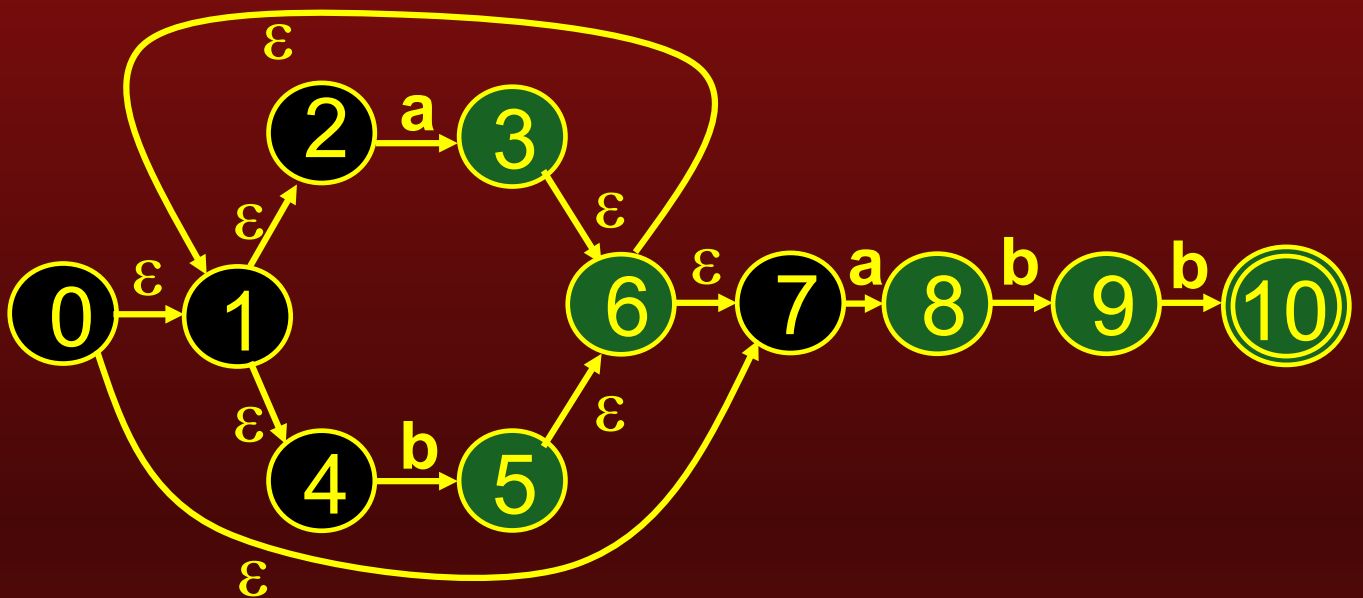
- The input symbol alphabet here is **{a,b}**.

20



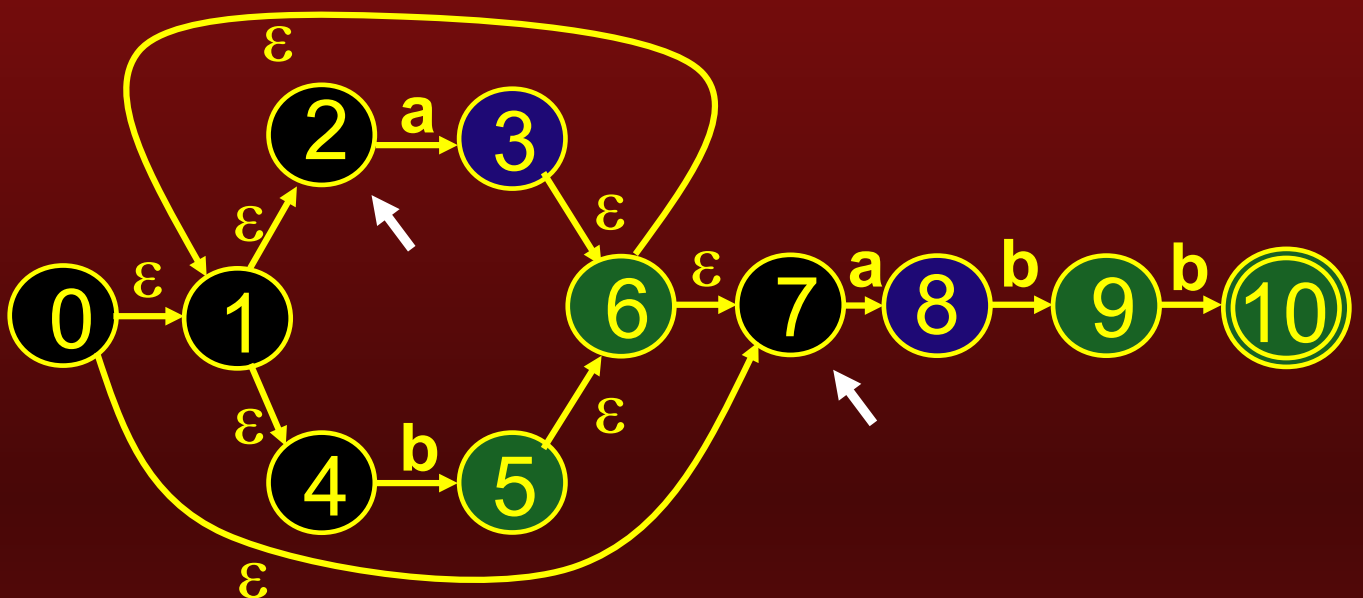
- The algorithm tells us to mark **A** and then compute $\epsilon\text{-closure}(\text{move}(A, \mathbf{a}))$

21



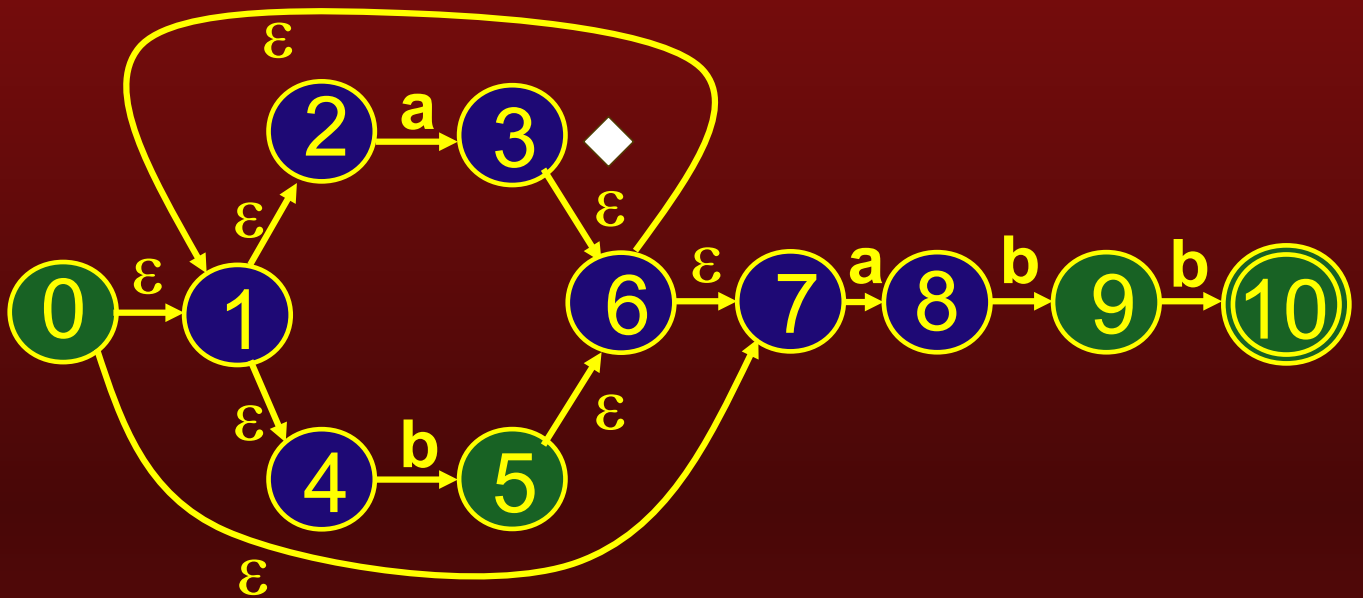
- $move(A, a)$, is the set of states of NFA that have transition on 'a' from members of A .

22



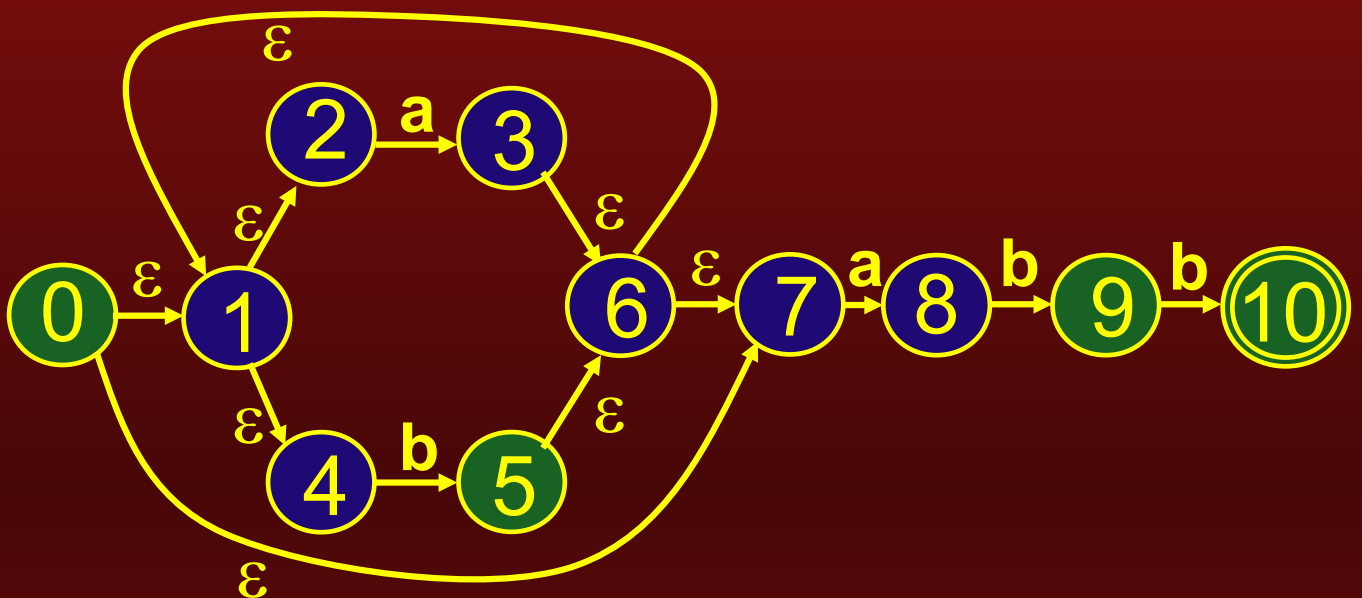
- Only 2 and 7 have such transition, to 3 and 8.

23



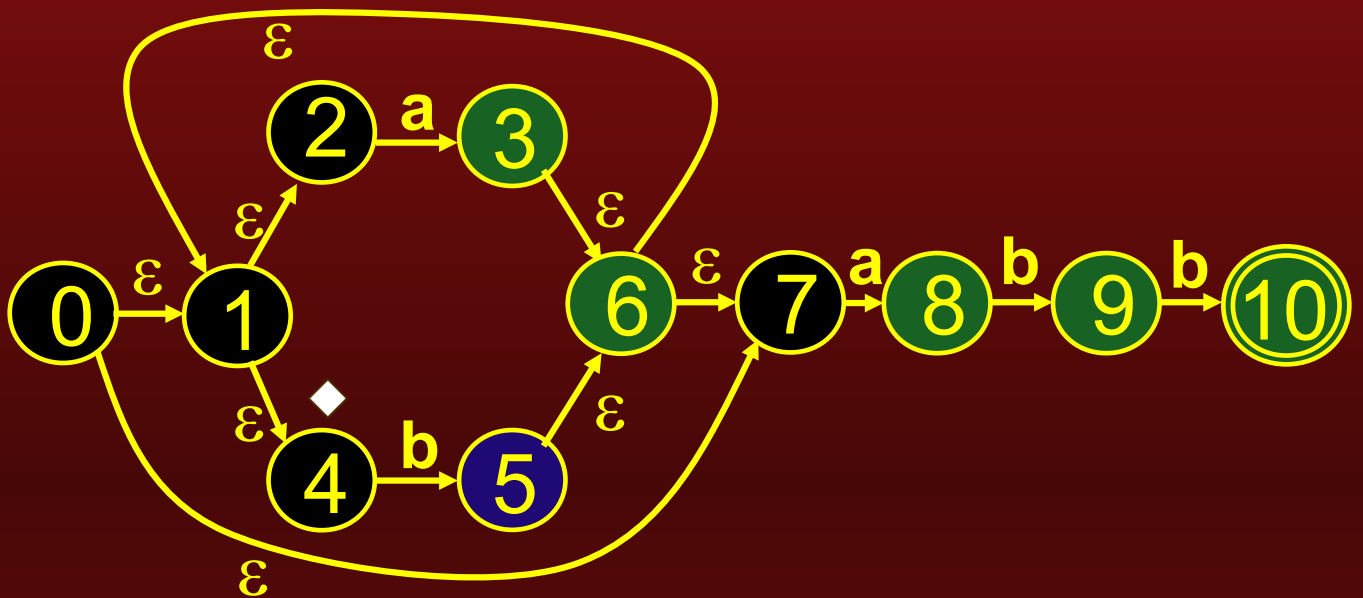
- So, $\varepsilon\text{-closure}(\text{move}(A, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$

24



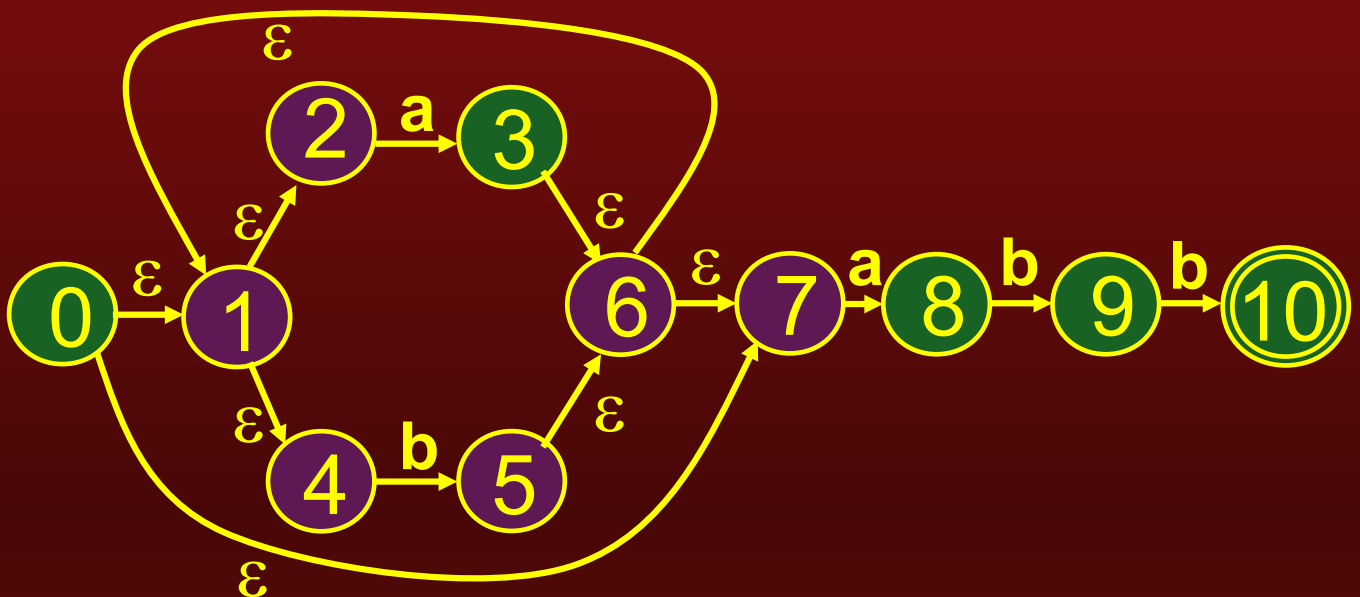
- Let $B = \{1, 2, 3, 4, 6, 7, 8\}$.
- Thus $Dtran[A, a] = B$

25



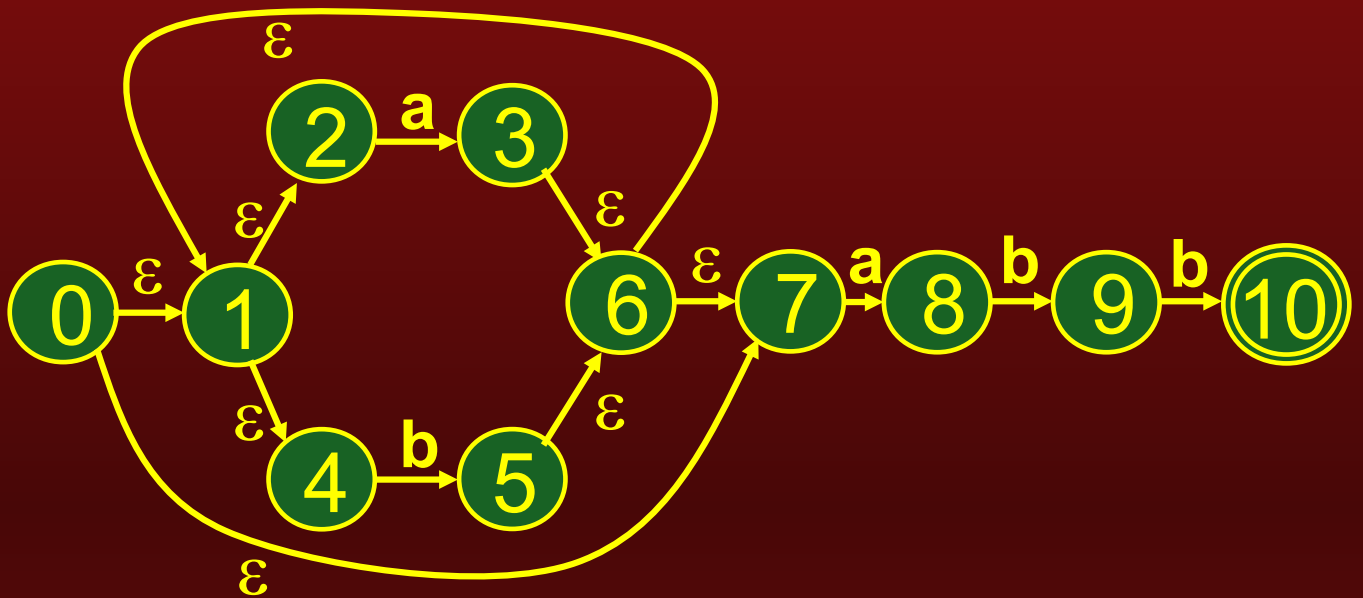
- For input **b**, among states in **A**, only **4** has transition on **b** to **5**

26



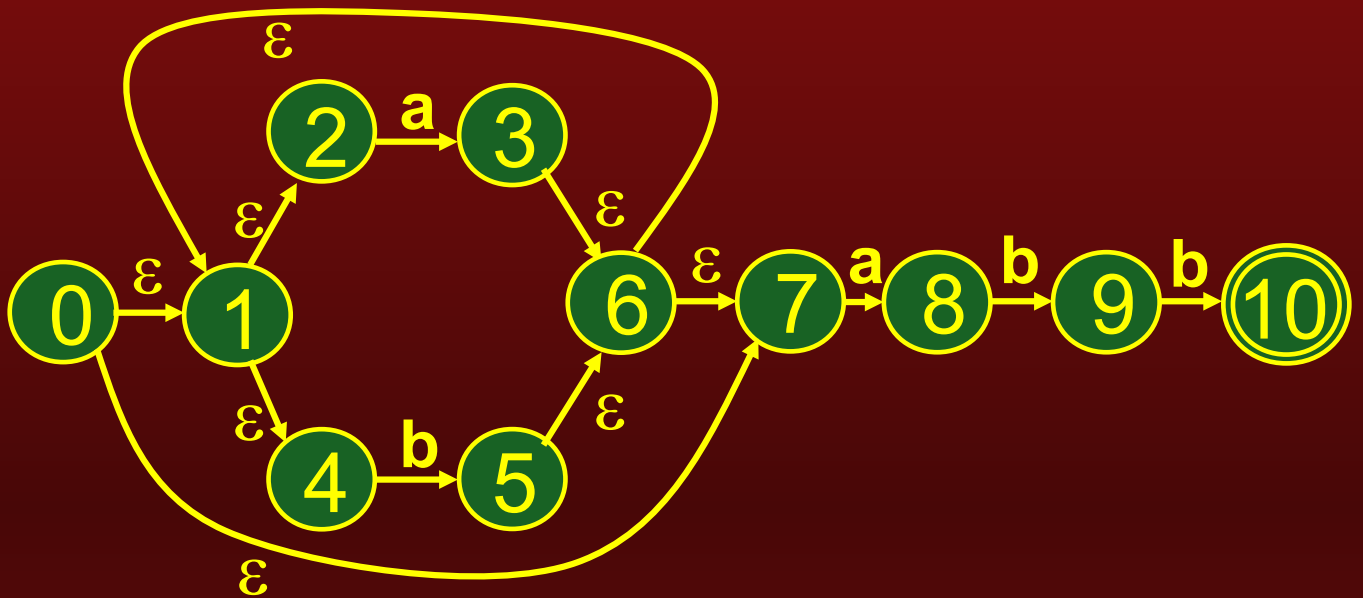
- $C = \varepsilon\text{-closure}(\{5\})$
 $= \{1, 2, 4, 5, 6, 7\}$
- Thus, $Dtran[A, b] = C$

27



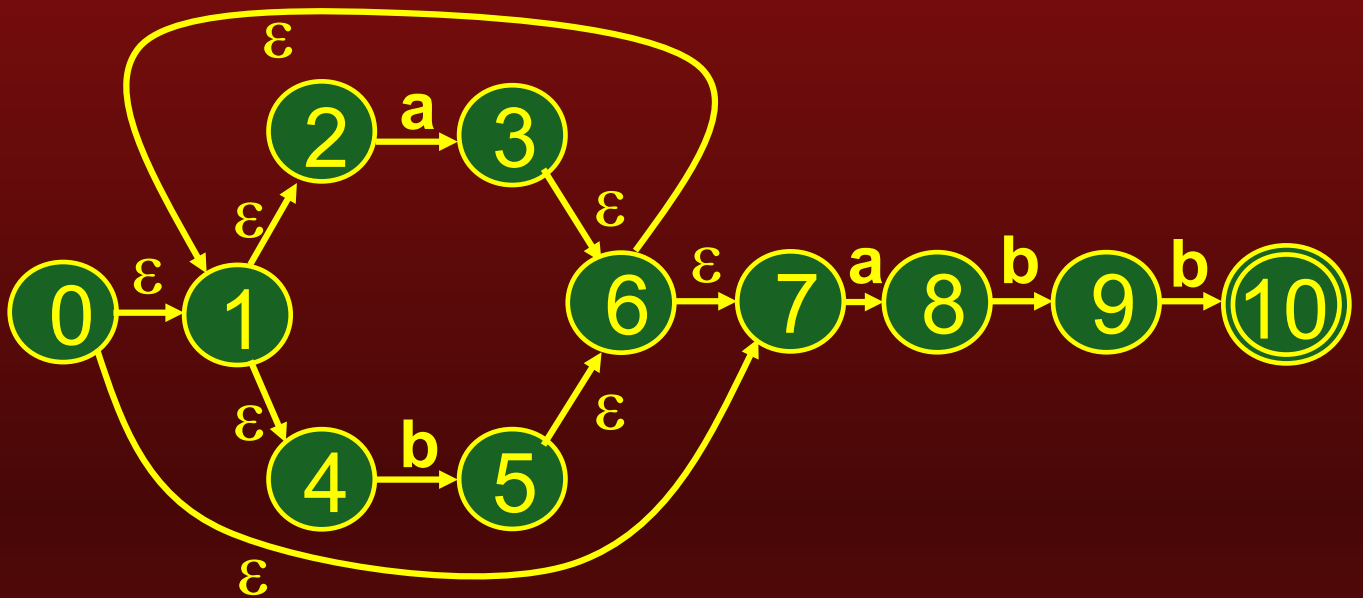
- We continue this process with the unmarked sets **B** and **C**

28



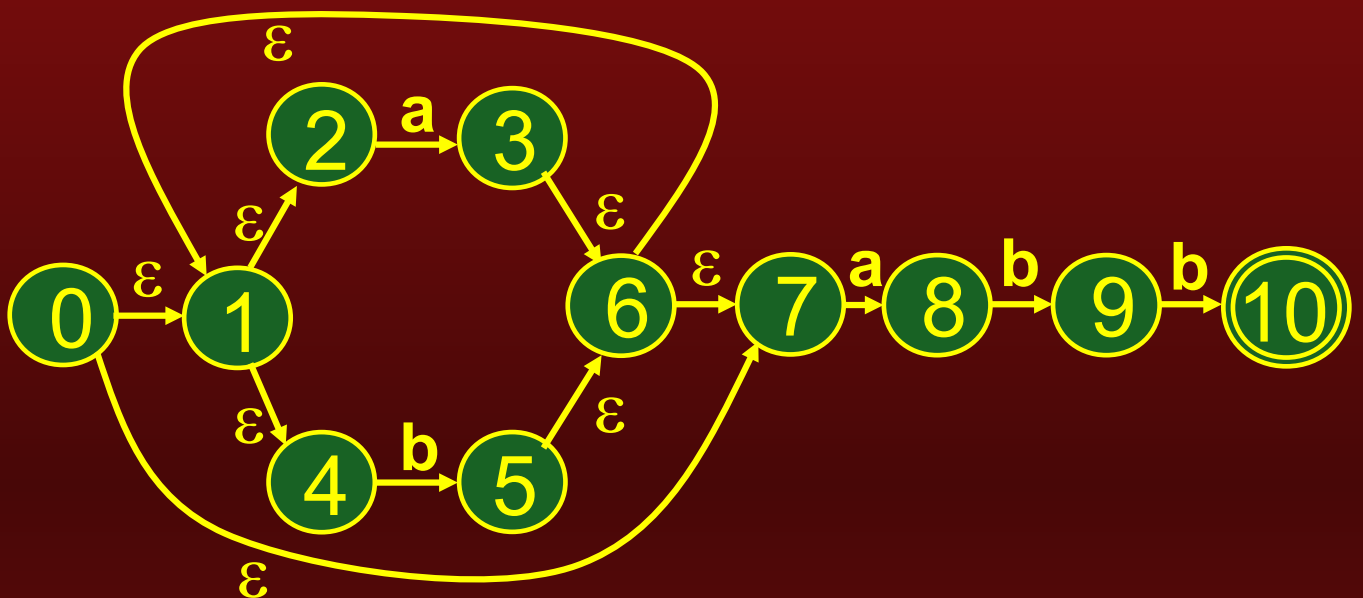
- i.e., $\epsilon\text{-closure}(\text{move}(B, a))$,
 $\epsilon\text{-closure}(\text{move}(B, b))$,

29



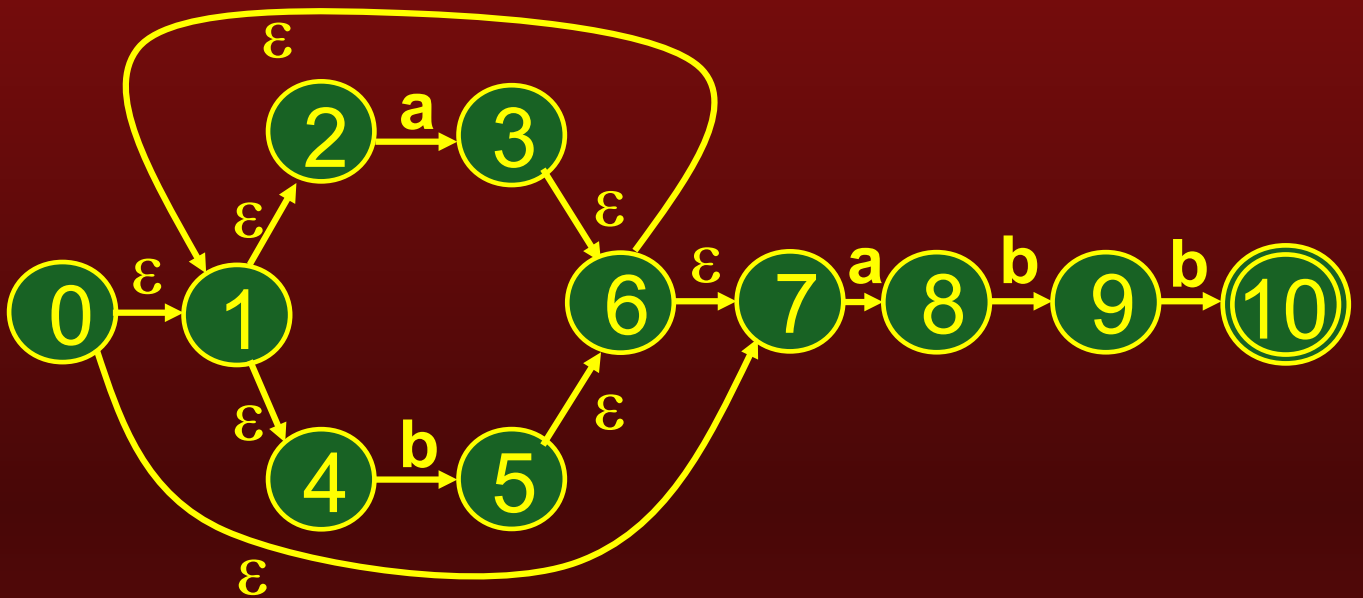
- ϵ -closure($move(C, a)$) and ϵ -closure($move(C, b)$)

30



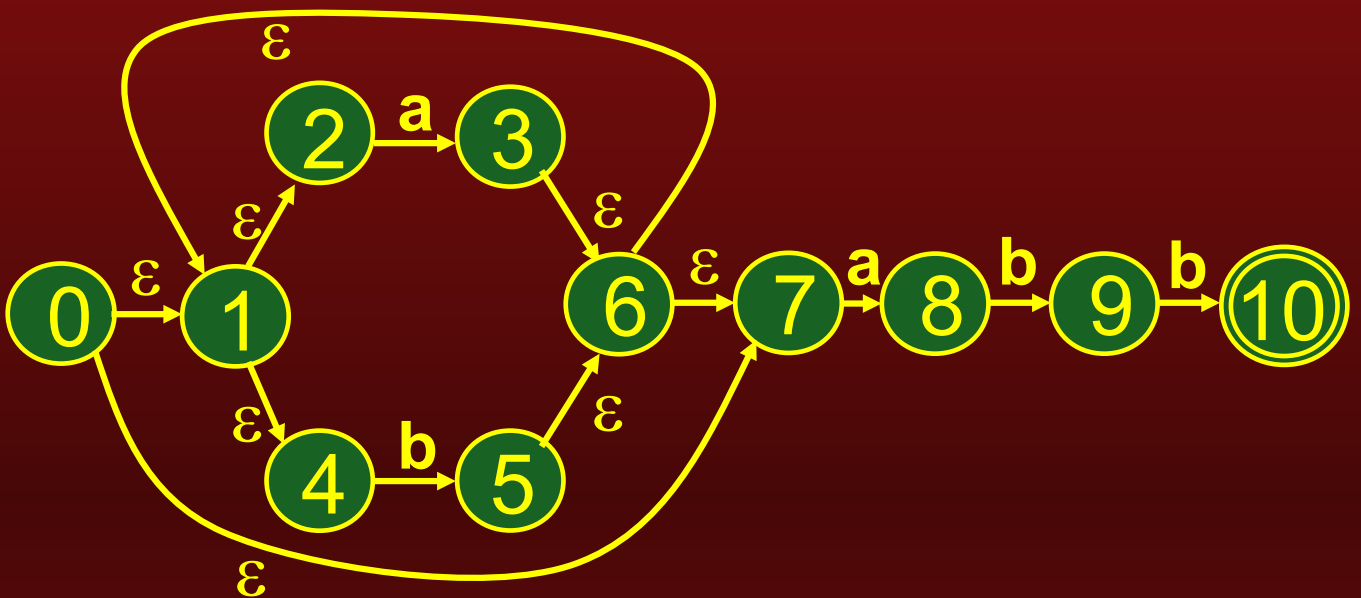
- Until all sets and states of DFA are marked.

31



- This is certain since there are *only* 2^{11} (!) different subsets of a set of 11 states.

32



- And a set, once marked, is marked forever.

33

Subset Construction

Eventually, the 5 sets are:

$$A=\{0,1,2,4,7\}$$

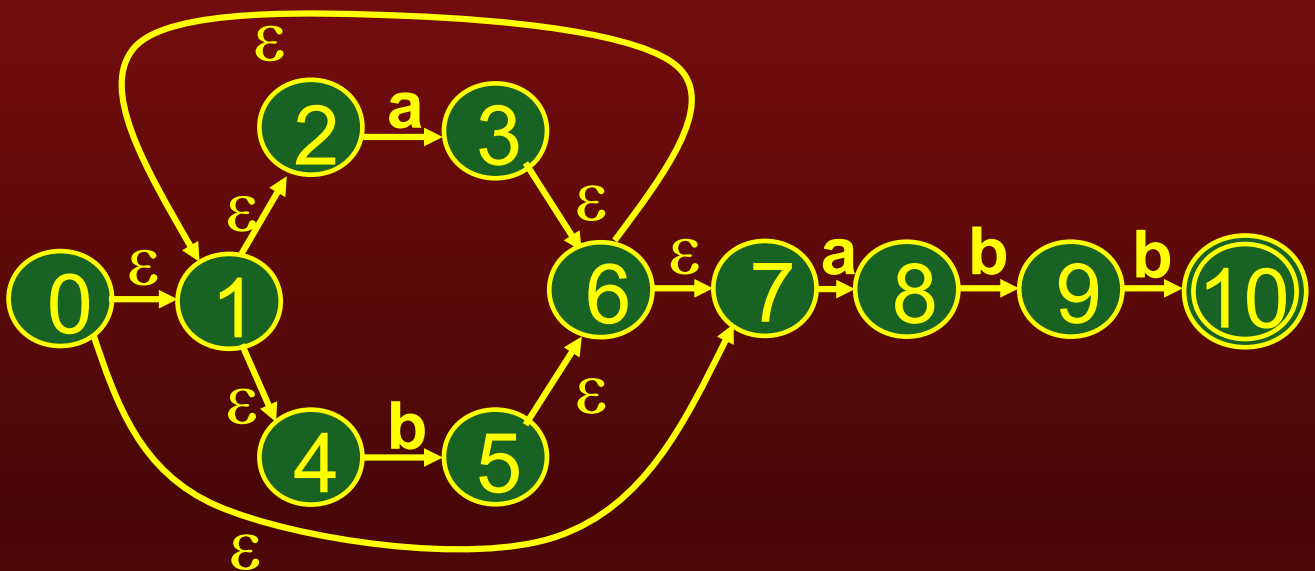
$$B=\{1,2,3,4,6,7,8\}$$

$$C=\{1,2,4,5,6,7\}$$

$$D=\{1,2,4,5,6,7,9\}$$

$$E=\{1,2,4,5,6,7,10\}$$

34



A is start state

$$A=\{0,1,2,4,7\}$$

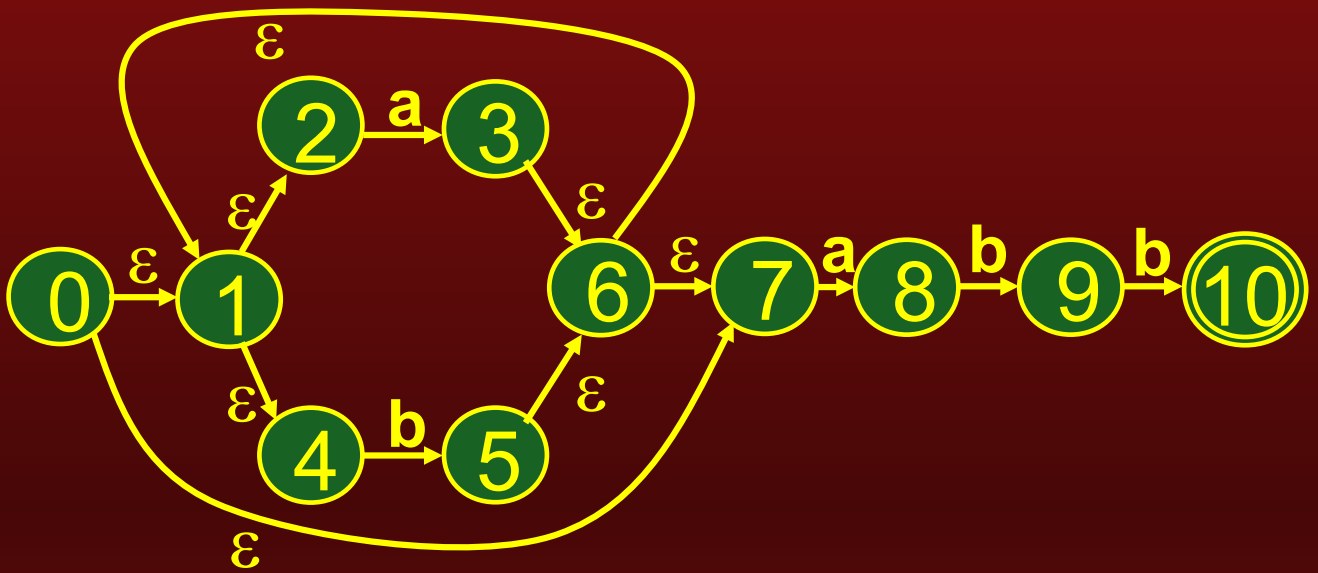
$$B=\{1,2,3,4,6,7,8\}$$

$$C=\{1,2,4,5,6,7\}$$

$$D=\{1,2,4,5,6,7,9\}$$

$$E=\{1,2,4,5,6,7,10\}$$

35



E is accepting state

$A = \{0, 1, 2, 4, 7\}$

$D = \{1, 2, 4, 5, 6, 7, 9\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

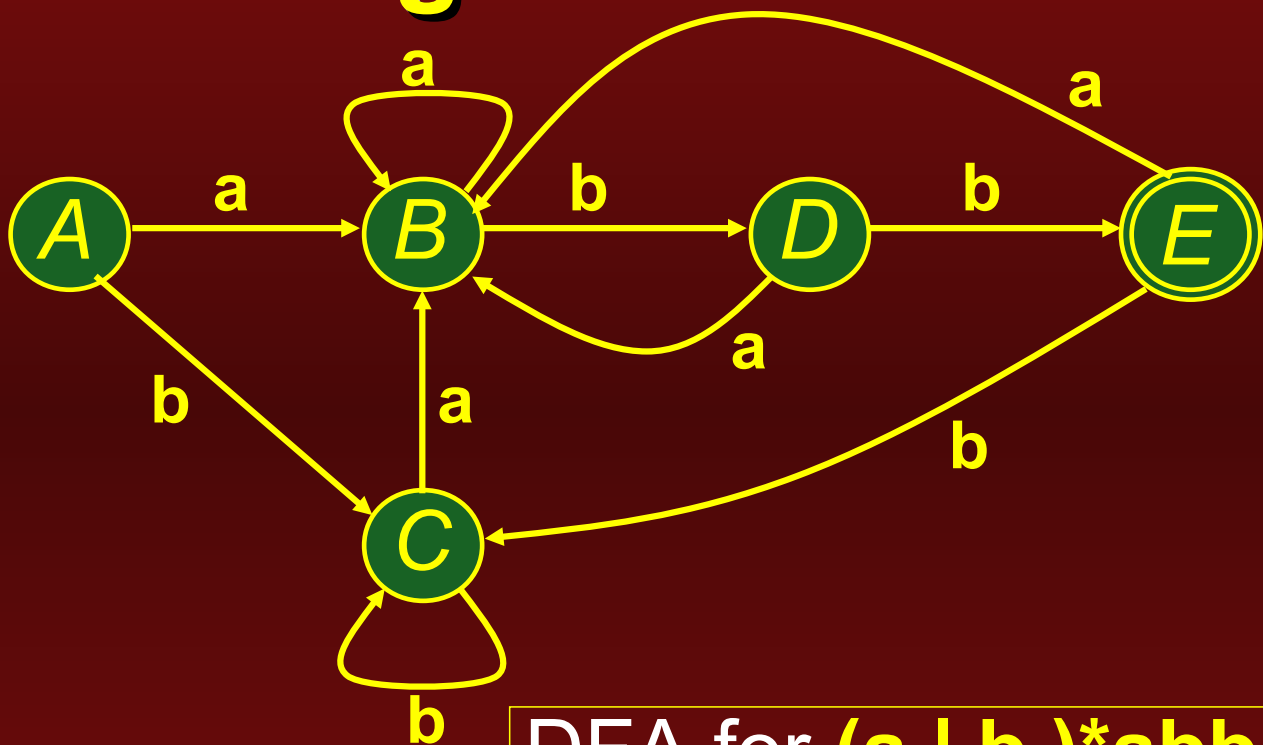
$E = \{1, 2, 4, 5, 6, 7, 10\}$

$C = \{1, 2, 4, 5, 6, 7\}$



36

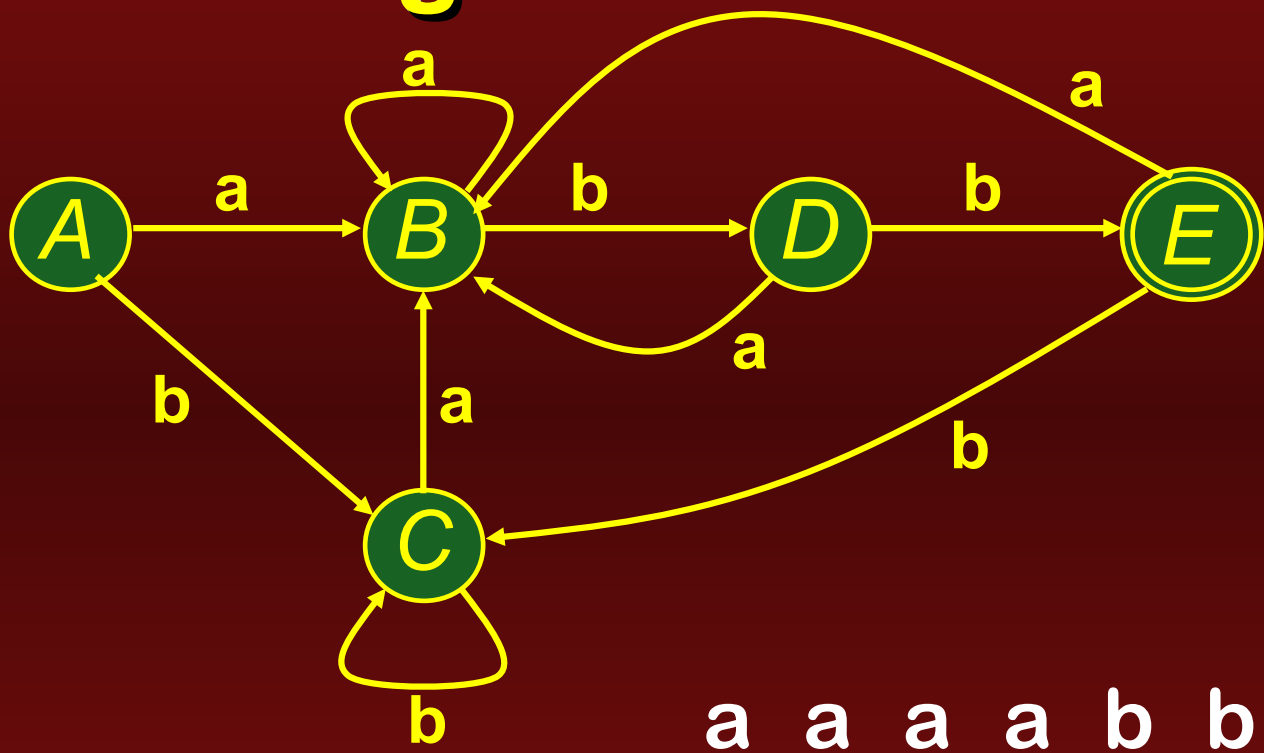
Resulting DFA



DFA for $(a \mid b)^*abb$

37

Resulting DFA



38

Final Transition Table

State	Input symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

39

Compiler Construction

Sohail Aslam

Lecture 9

DFA Minimization

- The generated DFA may have a **large** number of states.
- Hopcroft's algorithm: minimizes DFA states

DFA Minimization

- The generated DFA may have a **large** number of states.
- Hopcroft's algorithm: **minimizes** DFA states

3

DFA Minimization

- **Idea:** find groups of *equivalent states*.
- All transitions from states in one group G_1 go to states in the same group G_2

4

DFA Minimization

- Idea: find groups of *equivalent states*.
- All transitions from states in one group G_1 go to states in the same group G_2

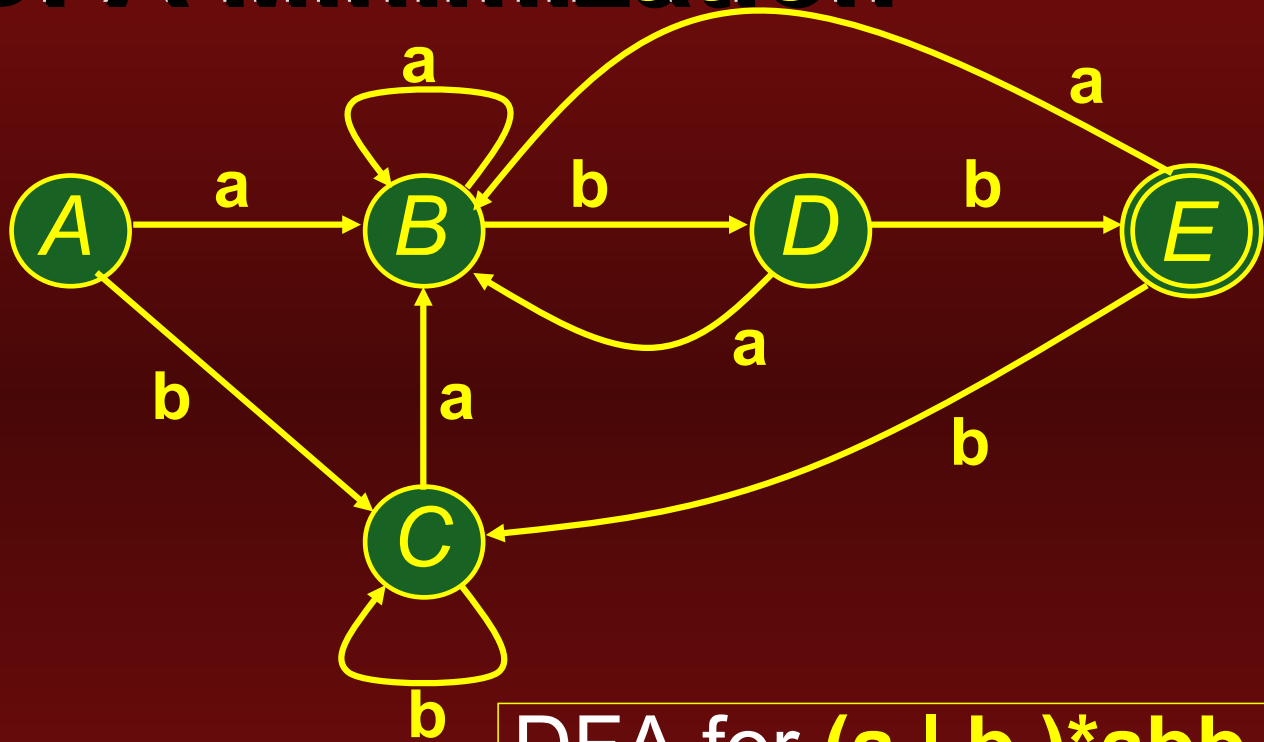
5

DFA Minimization

- Construct the minimized DFA such that there is **one state for each group** of states from the initial DFA.

6

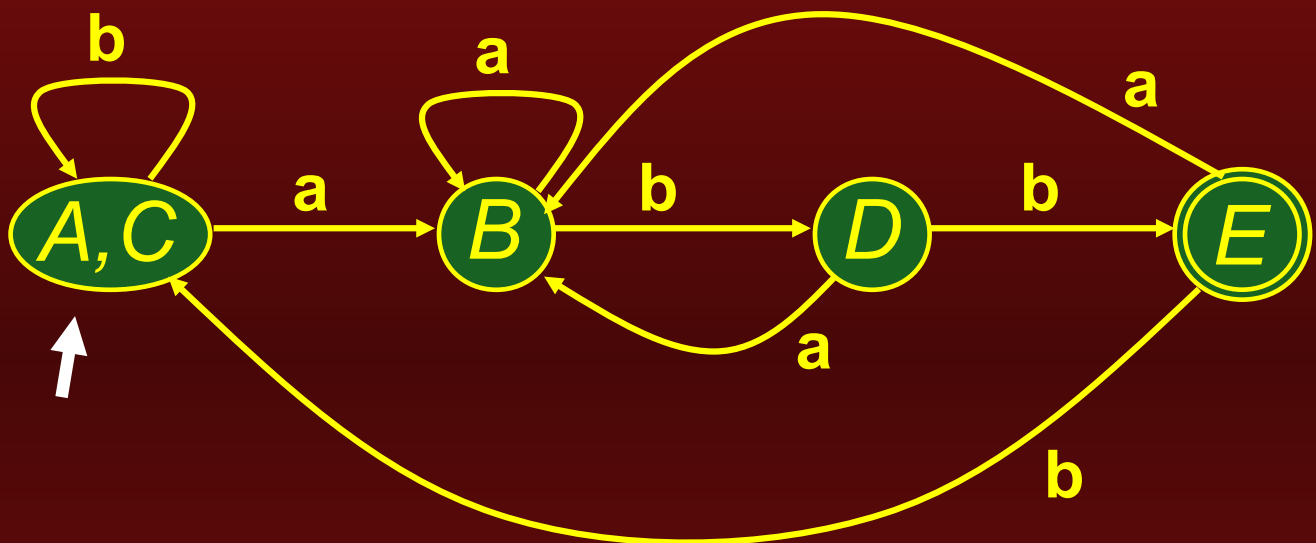
DFA Minimization



DFA for $(a \mid b)^*abb$

7

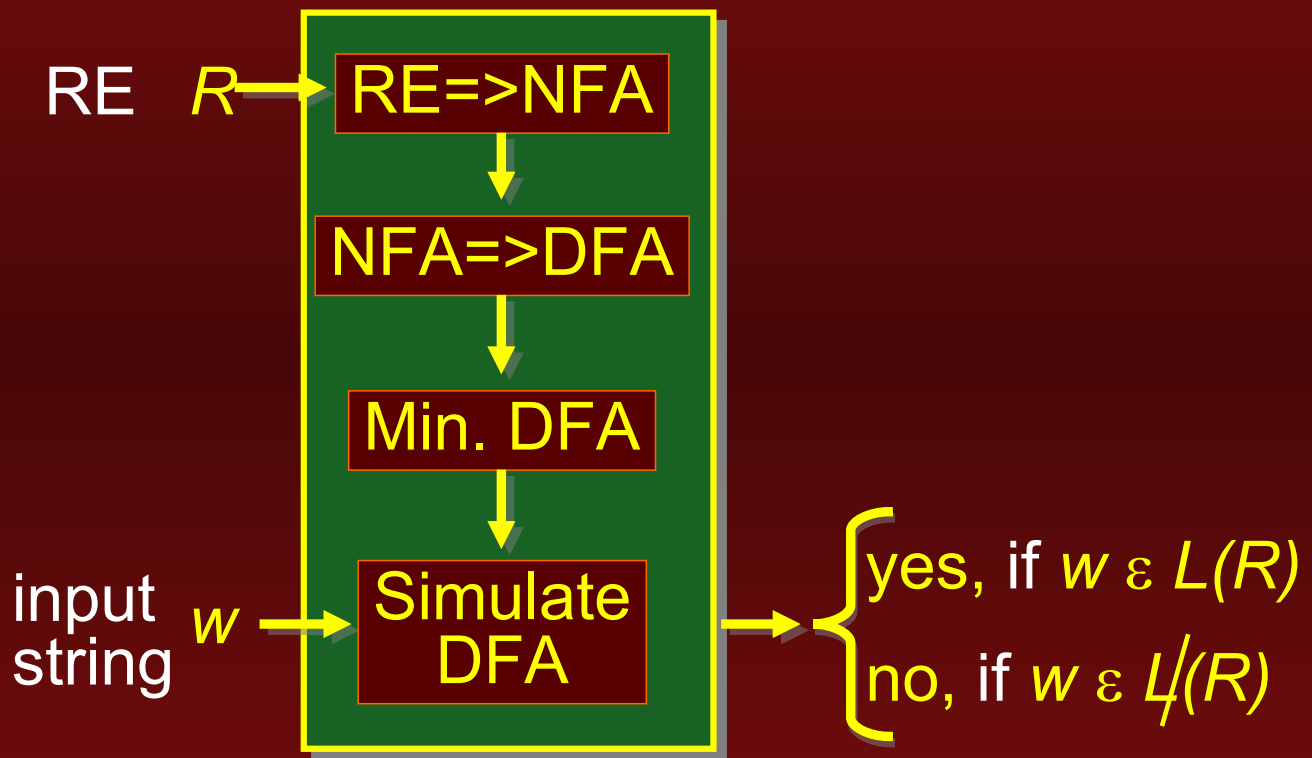
DFA Minimization



Minimized DFA for $(a \mid b)^*abb$

8

Optimized Acceptor



9

Lexical Analyzers

- Lexical analyzers (scanners) use the same mechanism
- but they:
 - Have multiple RE descriptions for multiple tokens
 - Have a character stream at the input

10

Lexical Analyzers

- Lexical analyzers (scanners) use the same mechanism
- but they:
 - Have **multiple RE** descriptions for multiple tokens
 - Have a character stream at the input

11

Lexical Analyzers

- Lexical analyzers (scanners) use the same mechanism
- but they:
 - Have **multiple RE** descriptions for multiple tokens
 - Have a **character stream** at the input

12

Lexical Analyzers

- **Return** a sequence of matching **tokens** at the output (or an error)
- Always return the longest matching token

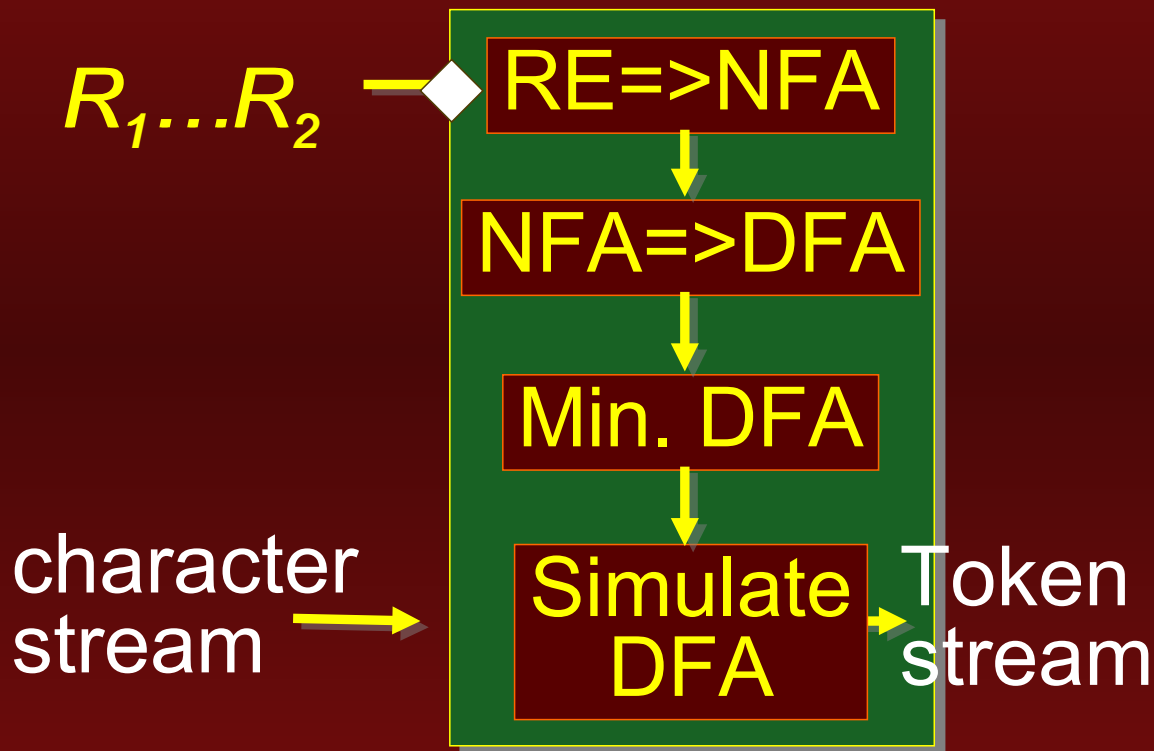
13

Lexical Analyzers

- **Return** a sequence of matching **tokens** at the output (or an error)
- Always return the longest matching **token**

14

Lexical Analyzers



15

Lexical Analyzer Generators

- The lexical analysis process can **automated**
- We only need to specify
 - Regular expressions for tokens
 - Rule priorities for multiple longest match cases

16

Lexical Analyzer Generators

- The lexical analysis process can **automated**
- We only need to specify
 - Regular expressions for tokens
 - Rule priorities for multiple longest match cases

17

Lexical Analyzer Generators

- **Flex**
generates lexical analyzer in C or C++
- **Jlex**
written in Java. Generates lexical analyzer in Java

18

Lexical Analyzer Generators

- **Flex**

generates lexical analyzer in C or C++

- **Jlex**

written in Java. Generates lexical analyzer in Java

19

Using Flex

- Provide a **specification file**

- Flex reads this file and produces C or C++ output file contains the scanner.

- The file consists of three sections

20

Using Flex

- Provide a **specification file**
- Flex reads this file and produces **C or C++** output file contains the scanner.
- The file consists of three sections

21

Using Flex

- Provide a **specification file**
- Flex reads this file and produces **C or C++** output file contains the scanner.
- The file consists of **three sections**

22

Flex Specification File

- 1 { C or C++ and flex definitions

23

Flex Specification File

- 1 { C or C++ and flex definitions
 - 2 { token definitions and actions
- %%

24

Flex Specification File

- 1 { C or C++ and flex definitions
%%
- 2 { token definitions and actions
%%
- 3 { user code

25

Specification File `lex.1`

```
%{  
#include "tokdefs.h"  
%}  
D      [0-9]  
L      [a-zA-Z_]  
id     {L} ({L}|{D})*  
%%  
"void" {return (TOK_VOID) ;}  
"int"  {return (TOK_INT) ;}  
"if"   {return (TOK_IF) ;}
```

26

Specification File lex.1

```
"else" {return (TOK_ELSE);}  
"while" {return (TOK_WHILE);}  
"<=" {return (TOK_LE);}  
">=" {return (TOK_GE);}  
"==" {return (TOK_EQ);}  
"!=" {return (TOK_NE);}  
{D}+ {return (TOK_INT);}  
{id} {return (TOK_ID);}  
[\\n]|[\\t]|[ ] ;  
%%
```

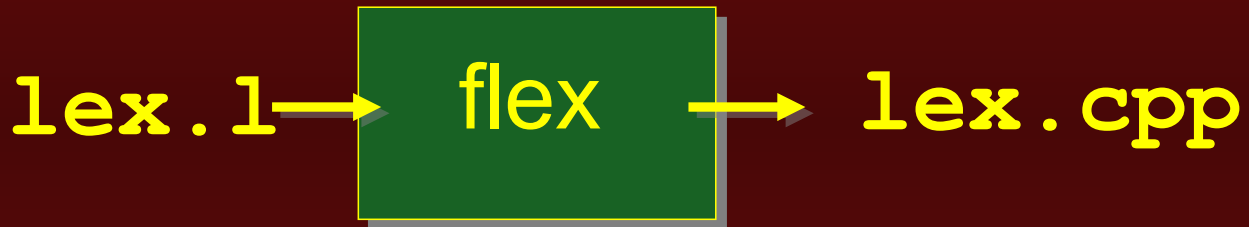
27

File tokdefs.h

```
#define TOK_VOID 1  
#define TOK_INT 2  
#define TOK_IF 3  
#define TOK_ELSE 4  
#define TOK_WHILE 5  
#define TOK_LE 6  
#define TOK_GE 7  
#define TOK_EQ 8  
#define TOK_NE 9  
#define TOK_INT 10  
#define TOK_ID 11
```

28

Invoking Flex



29

Using Generated Scanner

```
void main()  
{  
    FlexLexer lex;  
    int tc = lex.yylex();  
    while(tc != 0)  
        cout << tc << ", "  
            <<lex.YYText() << endl;  
    tc = lex.yylex();  
}
```

30

Creating Scanner EXE

```
flex lex.l  
g++ -c lex.cpp  
g++ -c main.cpp  
g++ -o lex.exe lex.o main.o  
  
lex <main.cpp
```

31

Compiler Construction

Sohail Aslam

Lecture 10

Using Generated Scanner

```
void main()  
{  
    FlexLexer lex;  
    int tc = lex.yylex();  
    while(tc != 0)  
        cout << tc << ", "  
            <<lex.YYText() << endl;  
    tc = lex.yylex();  
}
```

2

Input Tokenized

```
dos> .\lex < main.cpp  
259,void  
258,main  
283,(  
284,)  
285,{  
258,FlexLexer  
258,lex  
290,;  
260,int
```

3

Input Tokenized

```
258,tc  
266,=  
258,lex  
291,.  
258,yylex  
283,(  
284,)  
290,;  
263,while
```

4

Input Tokenized

```
283,(  
258,tc  
276,!=  
257,0  
284,)  
258,cout  
279,<<  
258,tc
```

5

Input Tokenized

```
279,<<  
292," ,"  
279,<<  
258,lex  
291, .  
258,YYText  
283, (  
284, )  
279,<<
```

6

Input Tokenized

```
258,endl  
290, ;  
258,tc  
266,=  
258,lex  
291, .  
258,yylex  
283, (  
284, )  
290, ;  
286, }
```

7

Flex Input for C++

```
/*
 * ISO C++ lexical analyzer.
 * Based on the ISO C++ draft standard of December
 * '96.
 */

%{
#include <ctype.h>
#include <stdio.h>
#include "tokdefs.h"

int lineno;

static int yywrap(void);
static void skip_until_eol(void);
static void skip_comment(void);
static int check_identifier(const char *);
%}
```

8

```
intsuffix ([uU][lL]?)|([lL][uU]?)
fracconst ([0-9]*\.[0-9]+)|([0-9]+\.)
exppart    [eE][+-]?[0-9]+
floatsuffix [fFlL]
chartext   ([^'])|(\.\.)
stringtext ([^"])|(\.\.)
%%
```

9

%%

```
"\n"      { ++lineno; }
[\\t\\f\\v\\r ]+  { /* Ignore whitespace. */ }

"/*"      { skip_comment(); }
"//"      { skip_until_eol(); }

 "{"      { return '{'; }
 "<%"     { return '{'; }
 "}"      { return '}'; }
 "%>"     { return '}'; }
 "["      { return '['; }
 "<:"     { return '['; }
```

10

```
 "]"      { return ']'; }
 ":>"     { return ']'; }
 "("      { return '('; }
 ")"      { return ')'; }
 ";"      { return ';'; }
 ":"      { return ':'; }
 "... "   { return ELLIPSIS; }
 "?"      { return '?'; }
 "::"     { return COLONCOLON; }
 "."      { return '.'; }
 ".*"     { return DOTSTAR; }
 "+"      { return '+'; }
 "-"      { return '-'; }
 "*"      { return '*'; }
 "/"      { return '/'; }
 "%"      { return '%'; }
 "^"      { return '^'; }
 "xor"    { return '^'; }
 "&"      { return '&'; }
 "bitand" { return '&'; }
```

11

```

"|"      { return '|'; }
"bitor"   { return '|'; }
"~"       { return '~'; }
"compl"   { return '~'; }
"!"       { return '!'; }
"not"     { return '!'; }
"="       { return '='; }
"<"      { return '<'; }
">"      { return '>'; }
"+="      { return ADDEQ; }
"-="      { return SUBEQ; }
"*="      { return MULEQ; }
"/="      { return DIVEQ; }
"%="      { return MODEQ; }
"^="      { return XOREQ; }
"xor_eq"  { return XOREQ; }
"&="     { return ANDEQ; }
"and_eq"  { return ANDEQ; }
"|="     { return OREQ; }
"or_eq"   { return OREQ; }

```

12

```

"<<"     { return SL; }
">>"     { return SR; }
"<<="    { return SLEQ; }
">>="    { return SREQ; }
"=="     { return EQ; }
"!="     { return NOTEQ; }
"not_eq" { return NOTEQ; }
"<="     { return LTEQ; }
">="     { return GTEQ; }
"&&"     { return ANDAND; }
"and"    { return ANDAND; }
"||"     { return OROR; }
"or"     { return OROR; }
"++"     { return PLUSPLUS; }
"--"     { return MINUSMINUS; }
","      { return ','; }
"->*"    { return ARROWSTAR; }
"->"     { return ARROW; }

```

13

```

"asm"      { return ASM; }
"auto"     { return AUTO; }
"bool"     { return BOOL; }
"break"    { return BREAK; }
"case"     { return CASE; }
"catch"    { return CATCH; }
"char"     { return CHAR; }
"class"    { return CLASS; }
"const"    { return CONST; }
"const_cast" { return CONST_CAST; }
"continue" { return CONTINUE; }
"default"  { return DEFAULT; }
"delete"   { return DELETE; }
"do"       { return DO; }
"double"   { return DOUBLE; }
"dynamic_cast" { return DYNAMIC_CAST; }
"else"     { return ELSE; }
"enum"     { return ENUM; }
"explicit" { return EXPLICIT; }
"export"   { return EXPORT; }

```

14

```

"extern"   { return EXTERN; }
"false"    { return FALSE; }
"float"    { return FLOAT; }
"for"      { return FOR; }
"friend"   { return FRIEND; }
"goto"     { return GOTO; }
"if"       { return IF; }
"inline"   { return INLINE; }
"int"      { return INT; }
"long"     { return LONG; }
"mutable"  { return MUTABLE; }
"namespace" { return NAMESPACE; }
"new"      { return NEW; }
"operator" { return OPERATOR; }
"private"  { return PRIVATE; }
"protected" { return PROTECTED; }
"public"   { return PUBLIC; }
"register" { return REGISTER; }
"reinterpret_cast" { return REINTERPRET_CAST; }
"return"   { return RETURN; }

```

15

```

"short"      { return SHORT; }
"signed"     { return SIGNED; }
"sizeof"     { return SIZEOF; }
"static"     { return STATIC; }
"static_cast" { return STATIC_CAST; }
"struct"     { return STRUCT; }
"switch"     { return SWITCH; }
"template"   { return TEMPLATE; }
"this"       { return THIS; }
"throw"      { return THROW; }
"true"       { return TRUE; }
"try"        { return TRY; }
"typedef"    { return TYPEDEF; }
"typeid"     { return TYPEID; }
"typename"   { return TYPENAME; }
"union"      { return UNION; }
"unsigned"   { return UNSIGNED; }
"using"      { return USING; }
"virtual"    { return VIRTUAL; }
"void"       { return VOID; }

```

16

```

"volatile"   { return VOLATILE; }
"wchar_t"    { return WCHAR_T; }
"while"      { return WHILE; }

```

```

[a-zA-Z_][a-zA-Z_0-9]*
{ return check_identifier(yytext); }

```

```

"0"[xX][0-9a-fA-F]+{intsuffix}? { return INTEGER; }
"0"[0-7]+{intsuffix}?           { return INTEGER; }
[0-9]+{intsuffix}?               { return INTEGER; }

```

17

```

{fracconst}{exppart}?{floatsuffix}? { return
    FLOATING; }
[0-9]+{exppart}{floatsuffix}? { return FLOATING; }

"'" {chartext}*'" { return CHARACTER; }
"L'" {chartext}*'" { return CHARACTER; }

"\\"" {stringtext}*"\\"" { return STRING; }
"L\"" {stringtext}*"\\"" { return STRING; }

```

18

```

. { fprintf(stderr,
    "%d: unexpected character `%c'\n", lineno,
    yytext[0]); }

%%

static int
yywrap(void)
{
    return 1;
}

```

19

```
static void
skip_comment(void)
{
    int c1, c2;

    c1 = input();
    c2 = input();

    while(c2 != EOF && !(c1 == '*' && c2 == '/'))
    {
        if (c1 == '\n')
            ++lineno;
        c1 = c2;
        c2 = input();
    }
}
```

20

```
static void
skip_until_eol(void)
{
    int c;

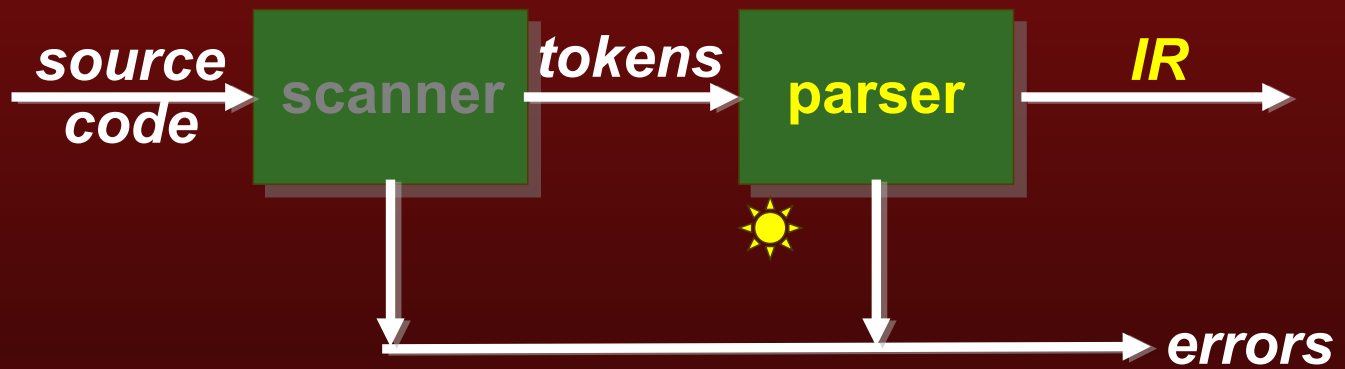
    while ((c = input()) != EOF && c != '\n')
        ;
    ++lineno;
}
```

21

```
static int
check_identifier(const char *s)
{
    /*
     * This function should check if `s' is a
     * typedef name or a class
     * name, or a enum name, ... etc. or
     * an identifier.
     */
    switch (s[0]) {
    case 'D': return TYPEDEF_NAME;
    case 'N': return NAMESPACE_NAME;
    case 'C': return CLASS_NAME;
    case 'E': return ENUM_NAME;
    case 'T': return TEMPLATE_NAME;
    }
    return IDENTIFIER;
}
```

Parsing

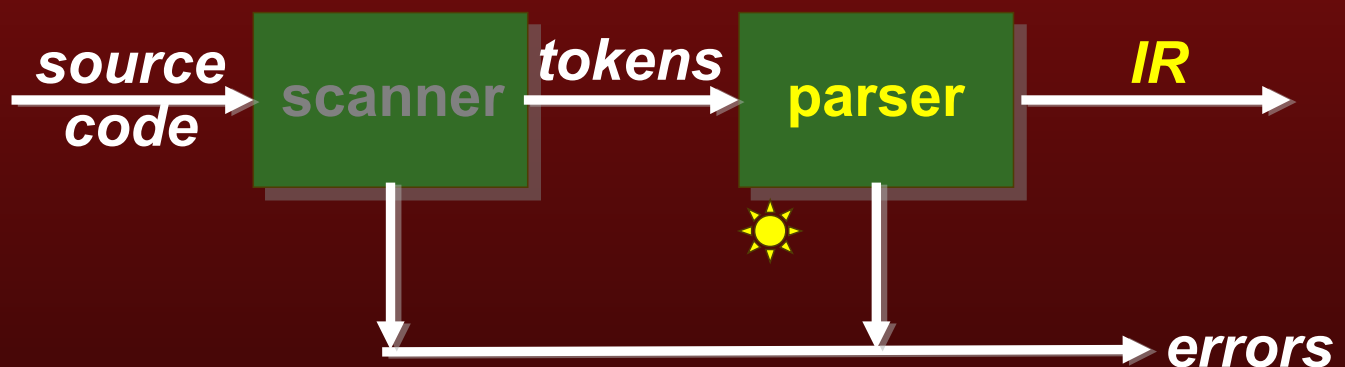
Front-End: Parser



- Checks the stream of words and their parts of speech for grammatical correctness

24

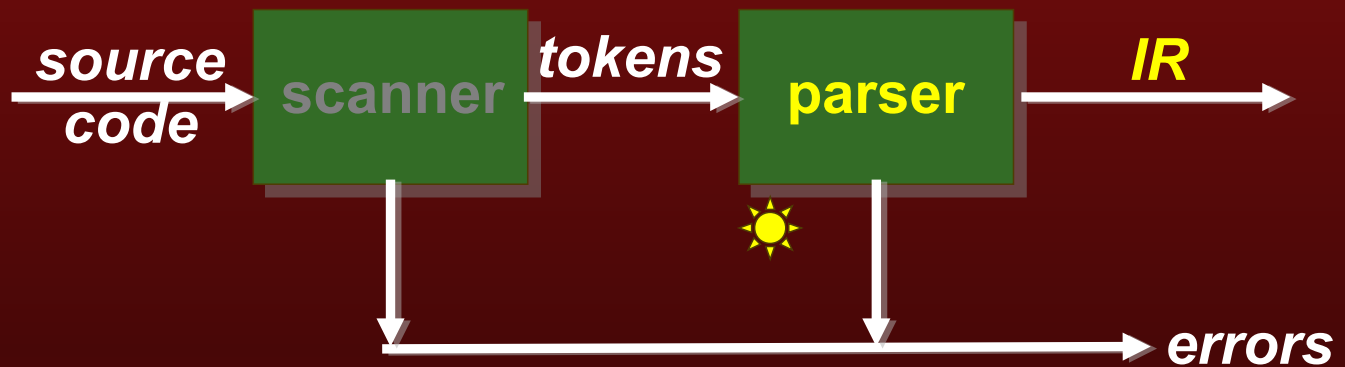
Front-End: Parser



- Determines if the input is syntactically well formed

25

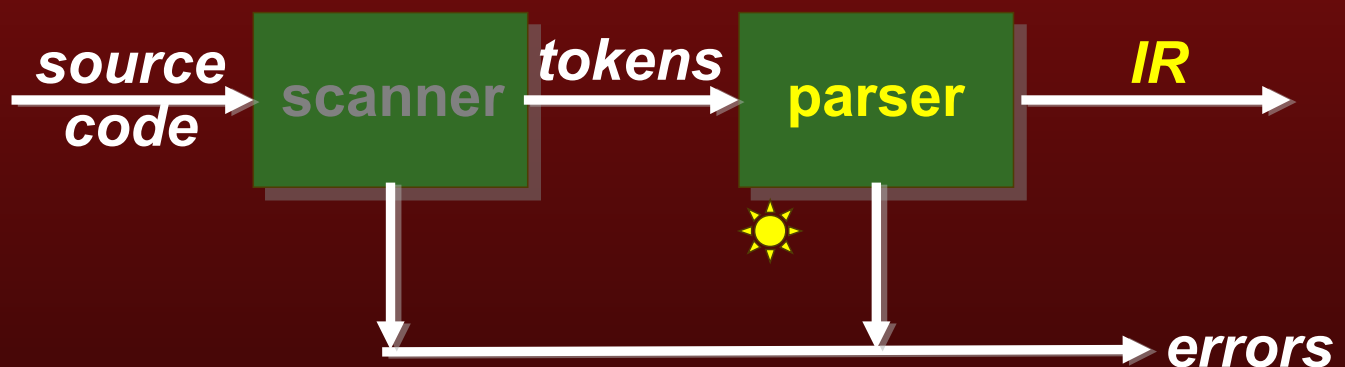
Front-End: Parser



- Guides context-sensitive (“**semantic**”) analysis (type checking)

26

Front-End: Parser



- Builds IR for source program

27

Syntactic Analysis

- Natural language analogy:
consider the sentence

He wrote the program

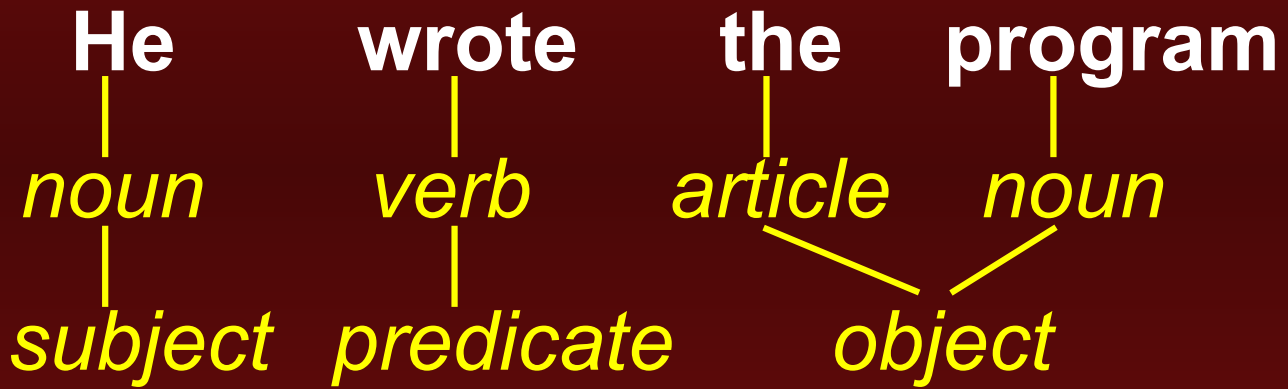
28

Syntactic Analysis

He wrote the program
noun verb article noun

29

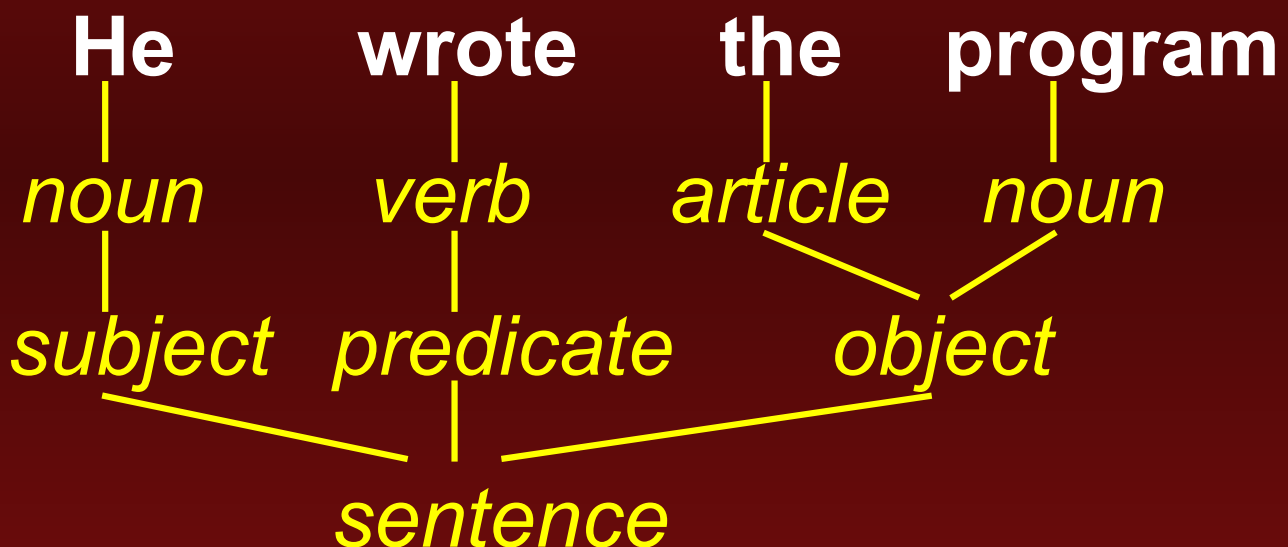
Syntactic Analysis



30

Syntactic Analysis

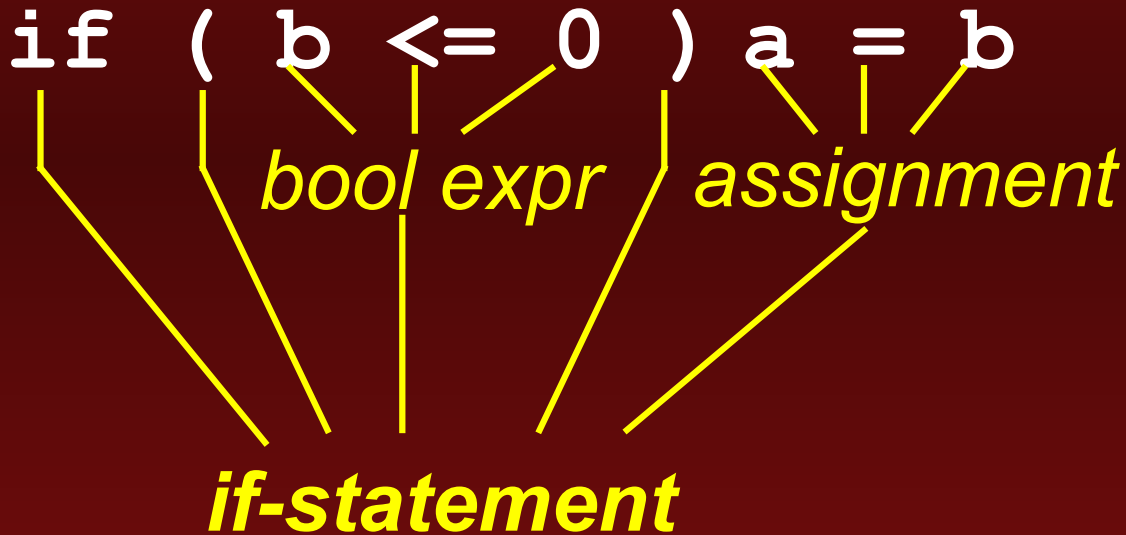
- Natural language analogy



31

Syntactic Analysis

- Programming language



32

Syntactic Analysis

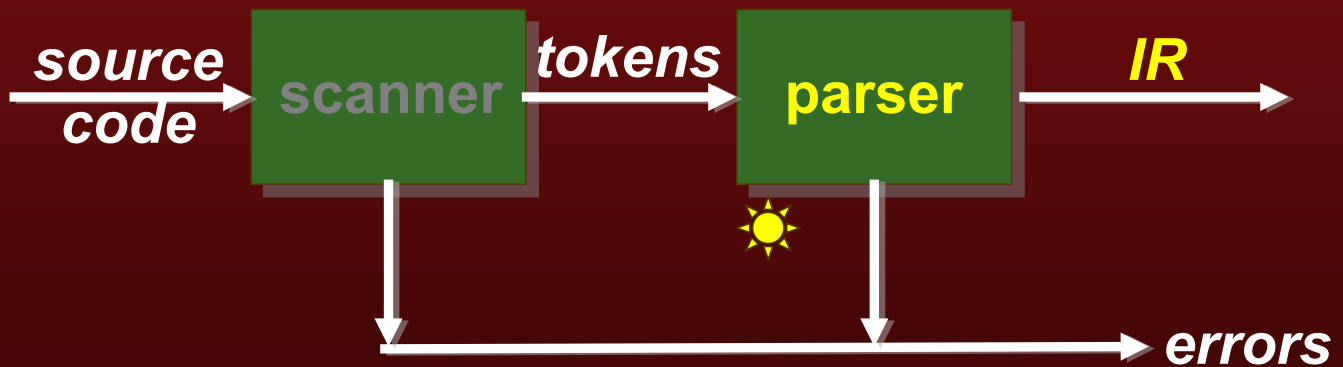
syntax errors

```
int* foo(int i, int j)
{
    for(k=0; i j; )
        fi( i > j )
        return j;
}
```

33

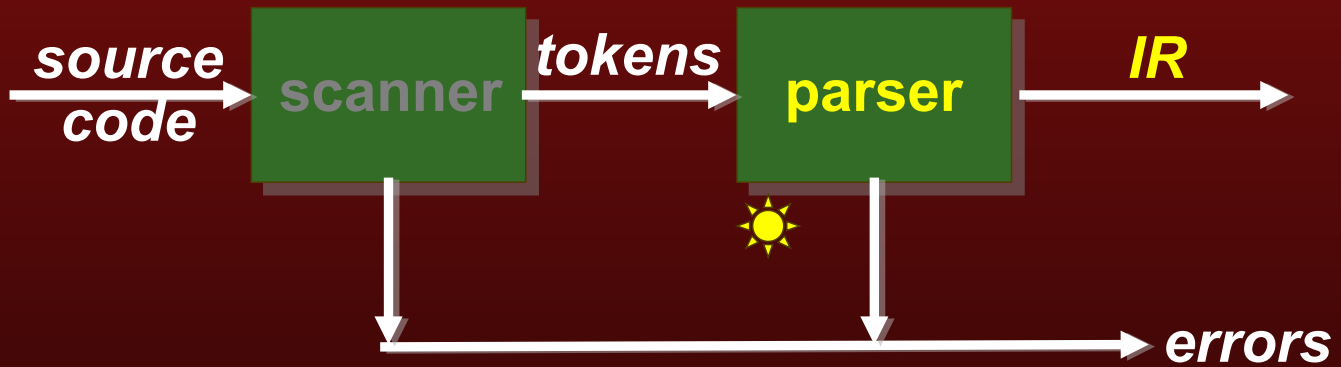
Parsing

Front-End: Parser



- Checks the stream of words and their parts of speech for grammatical correctness

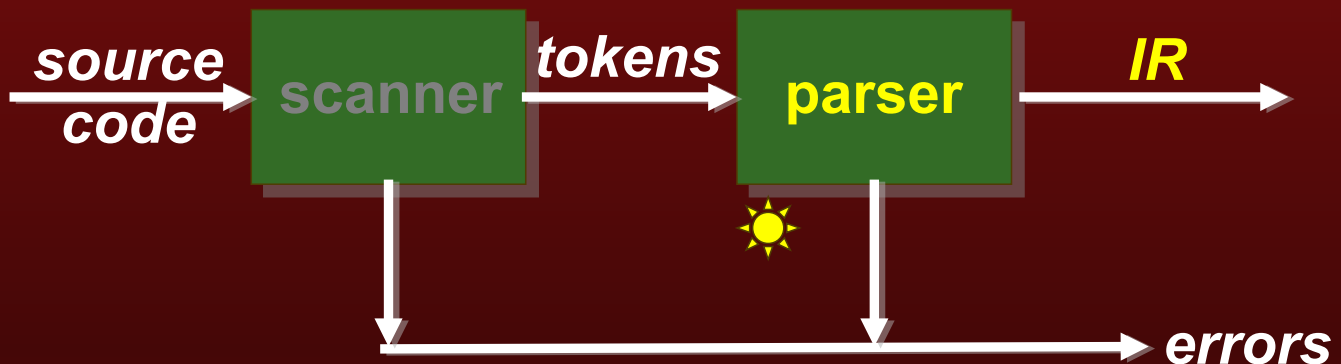
Front-End: Parser



- Determines if the input is syntactically well formed

3

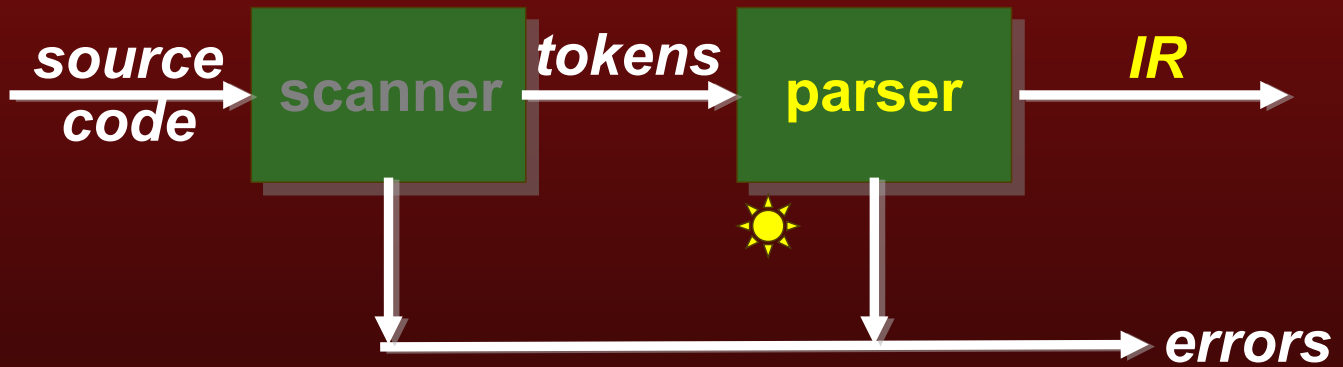
Front-End: Parser



- Guides context-sensitive (“**semantic**”) analysis (type checking)

4

Front-End: Parser



- Builds IR for source program

5

Syntactic Analysis

- Natural language analogy: consider the sentence

He wrote the program

6

Syntactic Analysis

He wrote the program
| | | |
noun *verb* *article* *noun*

7

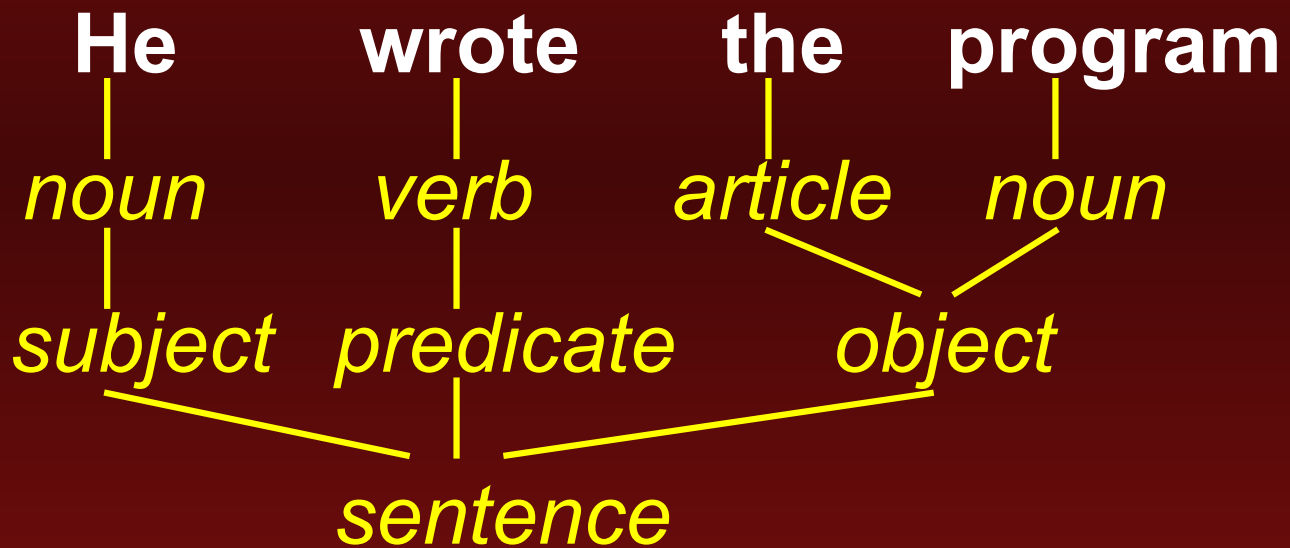
Syntactic Analysis

He wrote the program
| | | |
noun *verb* *article* *noun*
| | / \
subject *predicate* *object*

8

Syntactic Analysis

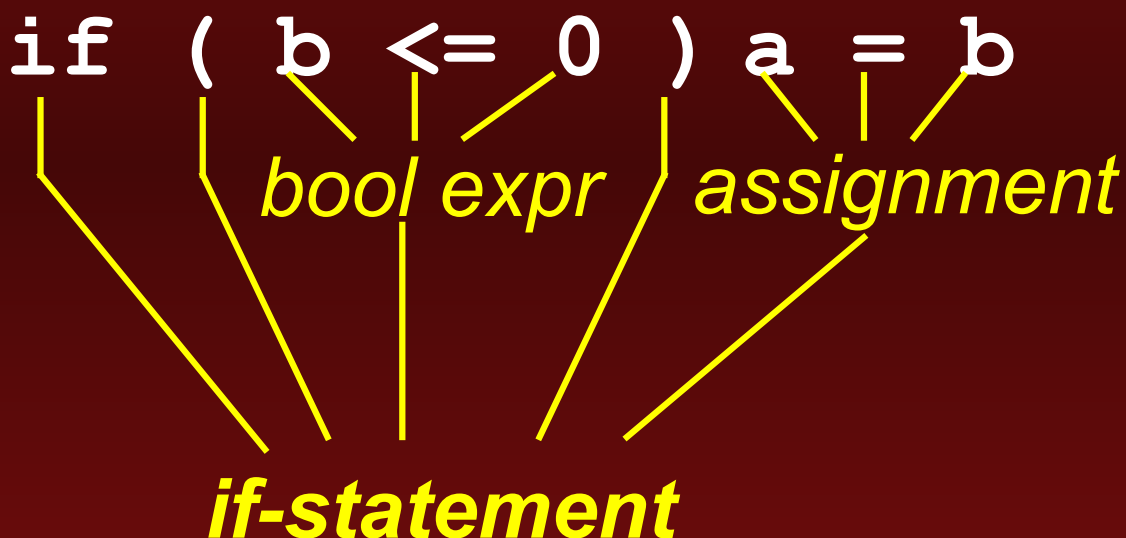
- Natural language analogy



9

Syntactic Analysis

- Programming language



10

Syntactic Analysis

syntax errors

```
int* foo(int i, int j)
{
    for(k=0; i j; )
        fi( i > j )
        return j;
}
```

11

Compiler Construction

Sohail Aslam

Lecture 11

Syntactic Analysis

```
int* foo(int i, int j))  
{  
    for(k=0; i j; )  
        fi( i > j )  
        return j;  
}
```

extra parenthesis

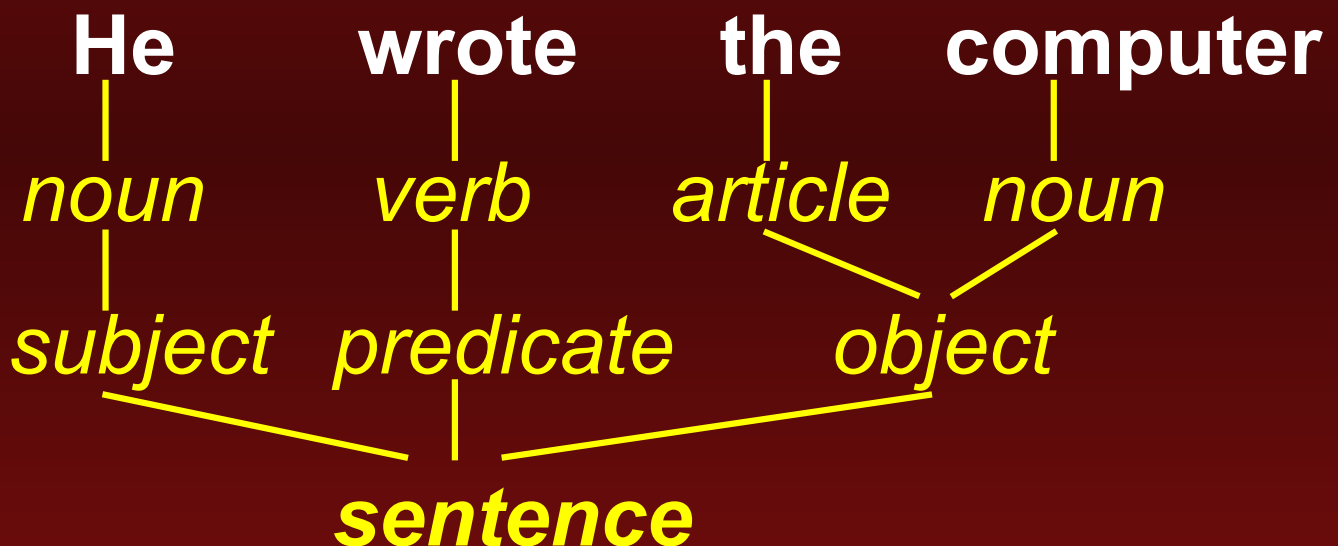
Missing expression

not a keyword

13

Semantic Analysis

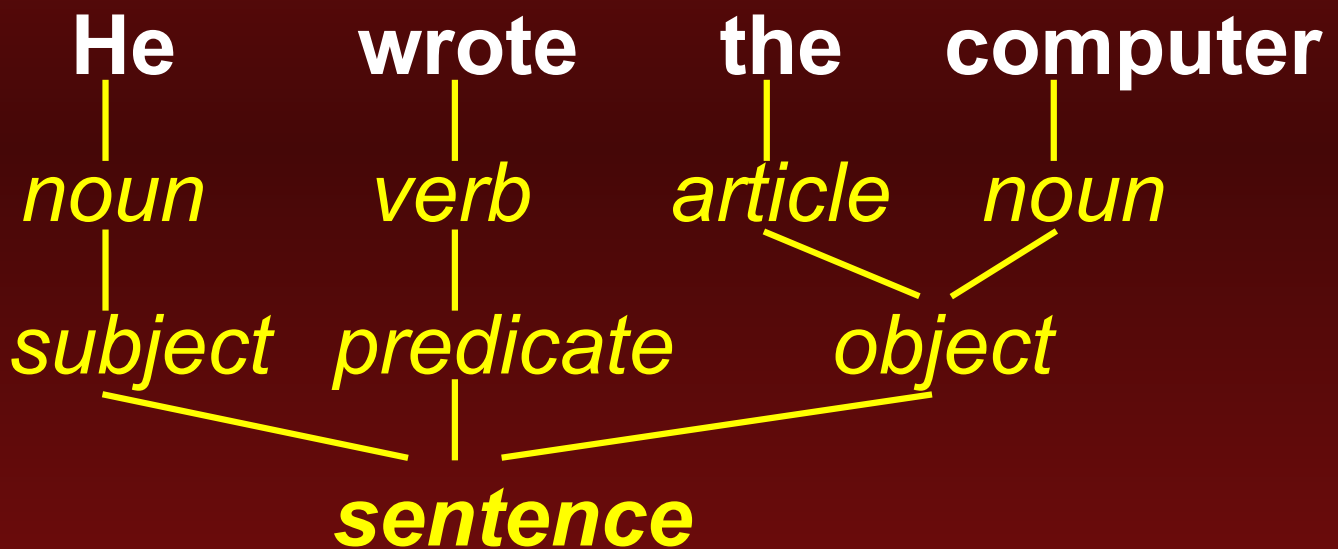
- Grammatically correct



14

Semantic Analysis

- semantically (meaning) wrong!



15

Semantic Analysis

```
int* foo(int i, int j)
{
    for(k=0; i < j; j++ )
        if( i < j-2 )
            sum = sum+i
    return sum;
}
```

undeclared var

return type mismatch

16

Role of the Parser

- Not all sequences of tokens are program.
- Parser must distinguish between valid and invalid sequences of tokens.

17

Role of the Parser

- Not all sequences of tokens are program.
- Parser must **distinguish** between **valid** and **invalid** sequences of tokens.

18

Role of the Parser

What we need

- An expressive way to describe the syntax
- An acceptor mechanism that determines if input token stream satisfies the syntax

19

Role of the Parser

What we need

- An expressive way to describe the syntax
- An acceptor mechanism that determines if input token stream satisfies the syntax

20

Role of the Parser

What we need

- An expressive way to describe the syntax
- An **acceptor mechanism** that determines if input token stream satisfies the syntax

21

Study of Parsing

- Parsing is the process of discovering a *derivation* for some sentence

22

Study of Parsing

- Mathematical model of syntax – a grammar G .
- Algorithm for testing membership in $L(G)$.

23

Study of Parsing

- Mathematical model of syntax – a grammar G .
- Algorithm for testing membership in $L(G)$.

24

Context Free Grammars

A CFG is a four tuple

$$G=(S,N,T,P)$$

- S is the *start symbol*
- N is a set of *non-terminals*
- T is a set of *terminals*
- P is a set of *productions*

25

Why Not Regular Expressions?

Reason:

regular languages do not have enough power to express syntax of programming languages.

26

Limitations of Regular Languages

- Finite automaton can't remember number of times it has visited a particular state

27

Example of CFG

- Context-free syntax is specified with a CFG

28

Example of CFG

- Example

SheepNoise → *SheepNoise* baa
 | baa

- This CFG defines the **set of noises** sheep make

29

Example of CFG

- We can use the *SheepNoise* grammar to create sentences
- We use the productions as rewriting rules

30

Example of CFG

SheepNoise → *SheepNoise* baa
 | baa

<i>Rule</i>	<i>Sentential Form</i>
-	<i>SheepNoise</i>
2	<u>baa</u>

31

Example of CFG

SheepNoise → *SheepNoise* baa
 | baa

<i>Rule</i>	<i>Sentential Form</i>
-	<i>SheepNoise</i>
1	<i>SheepNoise</i> <u>baa</u>
2	<u>baa</u> <u>baa</u>

32

Example of CFG

<i>Rule</i>	<i>Sentential Form</i>
-	<i>SheepNoise</i>
1	<i>SheepNoise</i> <u><i>baa</i></u>
1	<i>SheepNoise</i> <u><i>baa baa</i></u>
2	<u><i>baa</i></u> <u><i>baa</i></u> <u><i>baa</i></u>

And so on ...

33

Example of CFG

- While it is **cute**, this example quickly runs out intellectual steam
- To explore uses of CFGs, we need a more complex grammar

34

Example of CFG

- While it is **cute**, this example quickly runs out intellectual steam
- To explore uses of CFGs, we need a more complex grammar

35

More Useful Grammar

1	$expr \rightarrow expr\ op\ expr$
2	<u>num</u>
3	<u>id</u>
4	$op \rightarrow +$
5	$-$
6	$*$
7	$/$

36

Backus-Naur Form (BNF)

- Grammar rules in a similar form were first used in the description of the **Algol60** Language.

37

Backus-Naur Form (BNF)

- The notation was developed by **John Backus** and adapted by **Peter Naur** for the Algol60 report.
- Thus the term Backus-Naur Form (BNF)

38

Backus-Naur Form (BNF)

- The notation was developed by **John Backus** and adapted by **Peter Naur** for the Algol60 report.
- Thus the term **Backus-Naur Form (BNF)**

39

Derivation:

- Let us use the expression grammar to derive the sentence

$$x - 2 * y$$

40

Derivation: $x - 2 * y$

<i>Rule</i>	<i>Sentential Form</i>
-	<i>expr</i>
1	<i>expr op expr</i>
2	$\langle \text{id}, \underline{x} \rangle$ <i>op expr</i>
5	$\langle \text{id}, \underline{x} \rangle -$ <i>expr</i>
1	$\langle \text{id}, \underline{x} \rangle -$ <i>expr op expr</i>

41

Derivation: $x - 2 * y$

<i>Rule</i>	<i>Sentential Form</i>
2	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle$ <i>op expr</i>
6	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle *$ <i>expr</i>
3	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$

42

Derivation

- Such a process of rewrites is called a *derivation*.
- Process or discovering a derivations is called *parsing*

43

Derivation

- Such a process of rewrites is called a *derivation*.
- Process or discovering a derivations is called *parsing*

44

Derivation

We denote this derivation as:

$$\textit{expr} \rightarrow^* \underline{\textit{id}} - \underline{\textit{num}} * \underline{\textit{id}}$$

45

Derivations

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

46

Derivations

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

47

Derivations

- Two derivations are of interest
 1. **Leftmost** derivation
 2. **Rightmost** derivation

48

Derivations

- **Leftmost derivation:**
replace leftmost non-terminal (NT) at each step
- **Rightmost derivation:**
replace rightmost NT at each step

49

Derivations

- **Leftmost derivation:**
replace leftmost non-terminal (NT) at each step
- **Rightmost derivation:**
replace rightmost NT at each step

50

Derivations

- The example on the preceding slides was *leftmost derivation*
- There is also a *rightmost* derivation

51

Rightmost Derivation

<i>Rule</i>	<i>Sentential Form</i>
-	<i>expr</i>
1	<i>expr op expr</i>
3	<i>expr op</i> <id, <u>x</u> > 
6	<i>expr</i> * <id, <u>x</u> >
1	<i>expr op expr</i> * <id, <u>x</u> >

52

Derivation: $x - 2 * y$

<i>Rule</i>	<i>Sentential Form</i>
2	<i>expr op</i> $\langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{x} \rangle$
5	<i>expr</i> $- \langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{x} \rangle$
3	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$

53

Derivations

- In both cases we have

expr $\rightarrow^* \underline{\text{id}} - \underline{\text{num}} * \underline{\text{id}}$

54

Derivations

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

55

Derivations

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

56

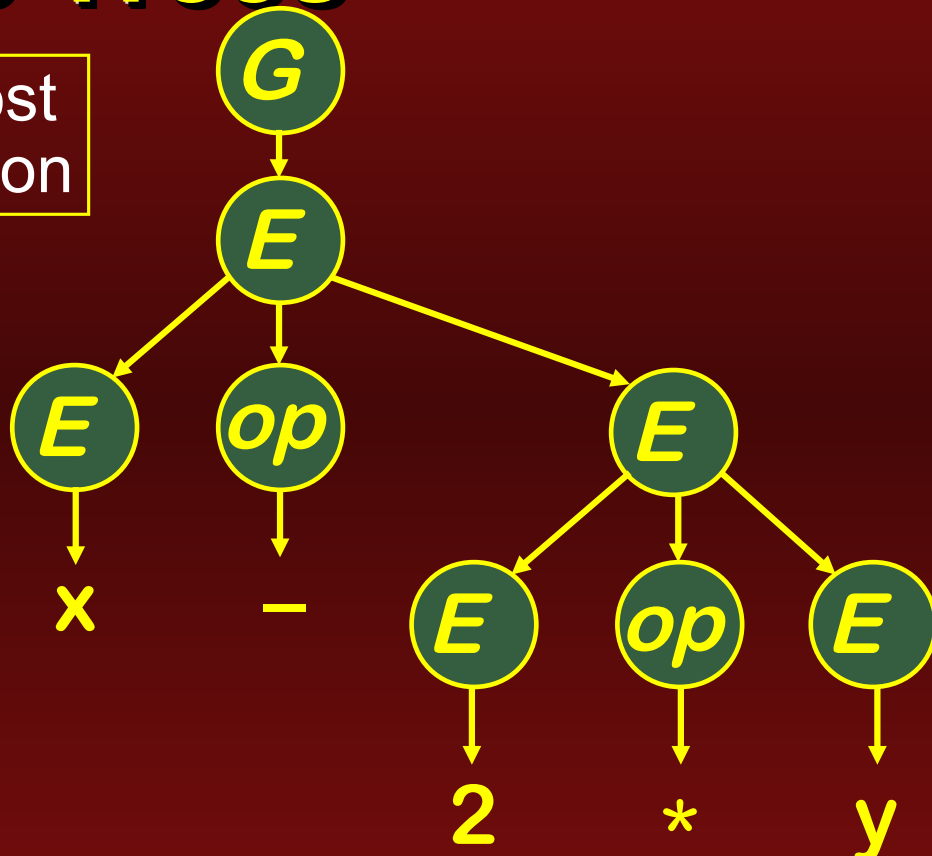
Compiler Construction

Sohail Aslam

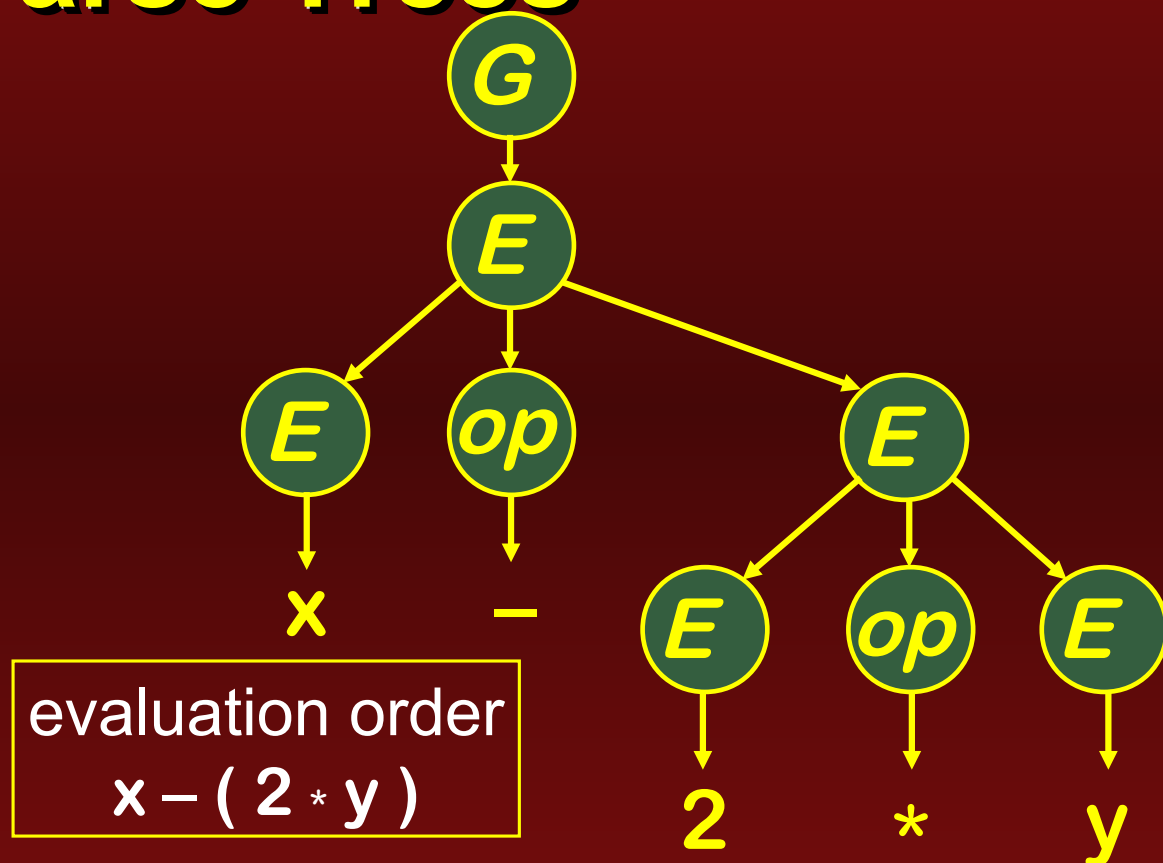
Lecture 12

Parse Trees

Leftmost
derivation

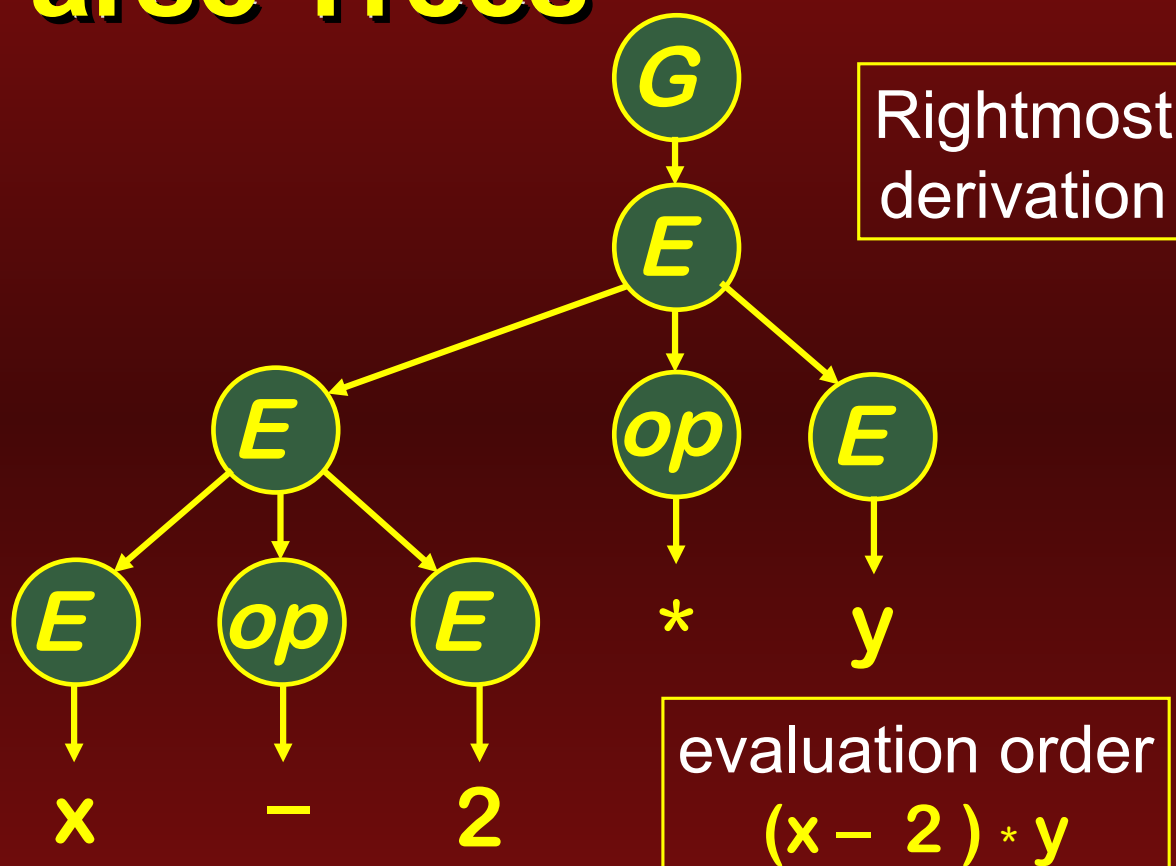


Parse Trees



3

Parse Trees



4

Precedence

- These two derivations point out a **problem** with the grammar
- It has no notion of *precedence*, or implied order of evaluation

5

Precedence

- These two derivations point out a **problem** with the grammar
- It has no notion of *precedence*, or implied order of evaluation

6

Precedence

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate corresponding part of grammar

7

Precedence

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate corresponding part of grammar

8

Precedence

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate corresponding part of grammar

9

Precedence

To add precedence

- Force parser to recognize **high** precedence subexpressions **first**.

10

Precedence

For **algebraic** expressions

- Multiplication and division,
first. (level one)
- Subtraction and addition,
next (level two)

11

Precedence

For **algebraic** expressions

- Multiplication and division,
first. (**level one**)
- Subtraction and addition,
next (level two)

12

Precedence

For **algebraic** expressions

- Multiplication and division,
first. (level one)
- Subtraction and addition,
next (level two)

13

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>expr</i> + <i>term</i>
3			<i>expr</i> – <i>term</i>
4			<i>term</i>
5	<i>term</i>	→	<i>term</i> * <i>factor</i>
6			<i>term</i> / <i>factor</i>
7			<i>factor</i>
8	<i>factor</i>	→	<u>number</u>
9			<u>ld</u>

14

level
two

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>

level
one

15

Precedence

This grammar is larger

- Takes more rewriting to reach some of the terminal symbols
- But it encodes expected precedence

16

Precedence

This grammar is **larger**

- Takes more rewriting to reach some of the terminal symbols
- But it encodes expected precedence

17

Precedence

This grammar is **larger**

- Takes more rewriting to reach some of the terminal symbols
- But it encodes expected precedence

18

Precedence

- Produces same parse tree under leftmost and rightmost derivations
- Let's see how it parses
 $x - 2 * y$

19

Precedence

- Produces same parse tree under leftmost and rightmost derivations
- Let's see how it parses
 $x - 2 * y$

20

Precedence

<i>Rule</i>	<i>Sentential Form</i>
-	<i>Goal</i>
1	<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>

21

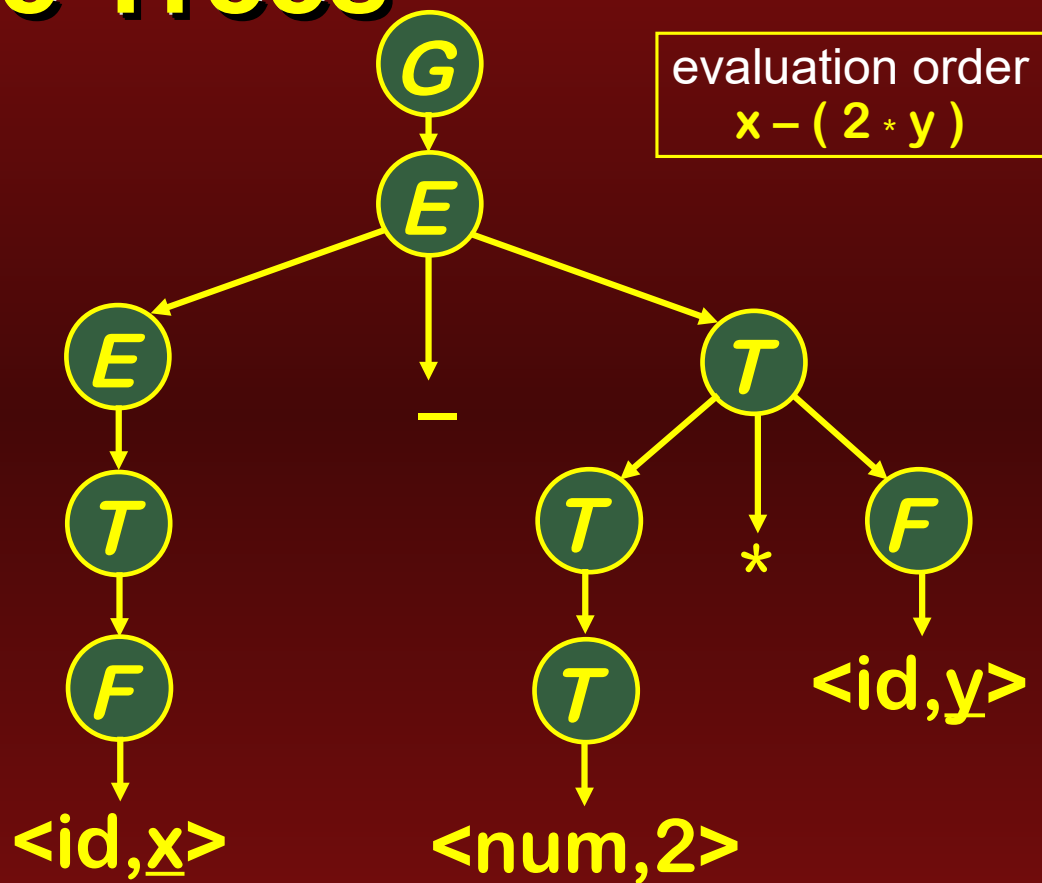
Derivations & Precedence

<i>Rule</i>	<i>Sentential Form</i>
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

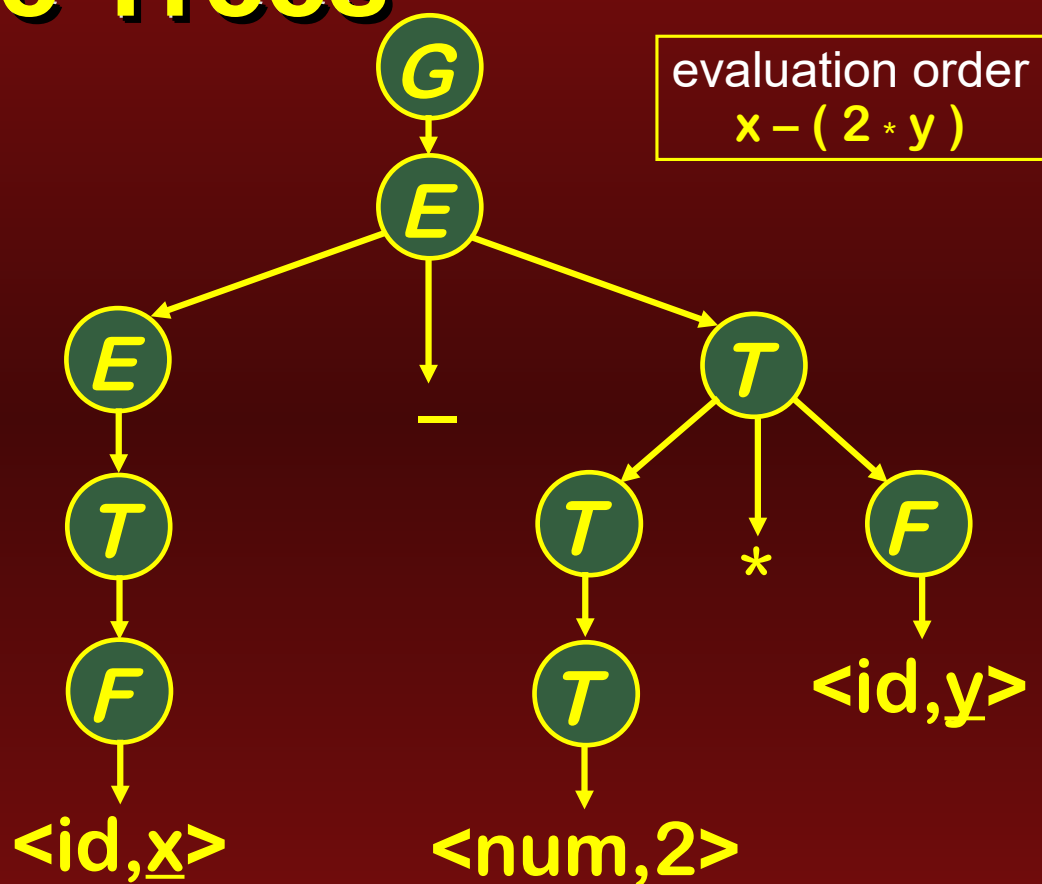
22

Parse Trees



23

Parse Trees



24

Precedence

- Both **leftmost** and **rightmost** derivations give the *same expression*
- Because the grammar directly encodes the desired precedence.

25

Precedence

- Both **leftmost** and **rightmost** derivations give the *same expression*
- Because the grammar directly encodes the desired precedence.

26

Ambiguous Grammars

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is *ambiguous*

27

Ambiguous Grammars

- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*

28

Ambiguous Grammars

- The **leftmost** and **rightmost** derivations for a sentential form may **differ**, even in an *unambiguous* grammar
- Let's consider the classic **if-then-else** example

29

Ambiguous Grammars

if-then-else problem

Stmt $\rightarrow$ **if** *Expr* **then** *Stmt*
 | **if** *Expr* **then** *Stmt* **else** *Stmt*
 | *... other stmts*

30

Ambiguous Grammars

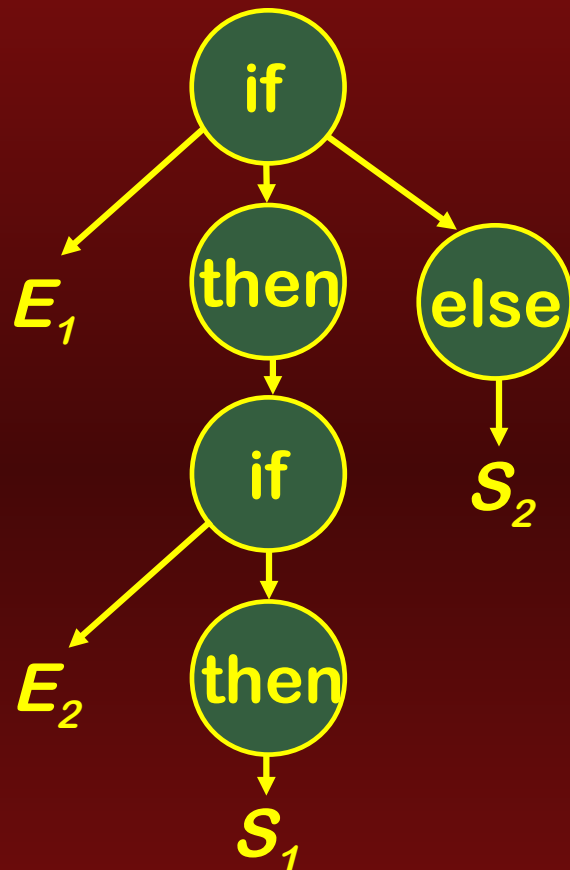
- The following sentential form has **two** derivations:

if E_1 then if E_2 then S else S_2

31

Ambiguity

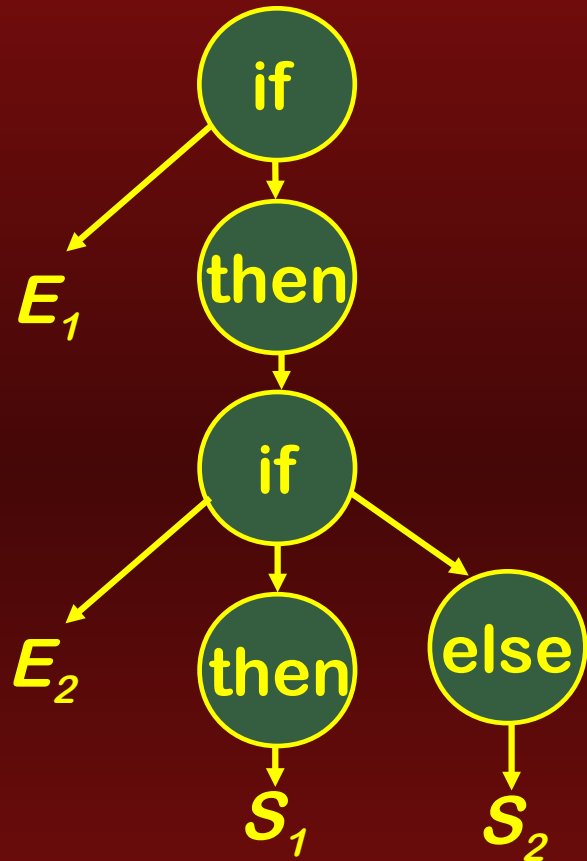
Production 1,
then
Production 2:
if E_1 **then**
 if E_2 **then** S_1
else S_2



32

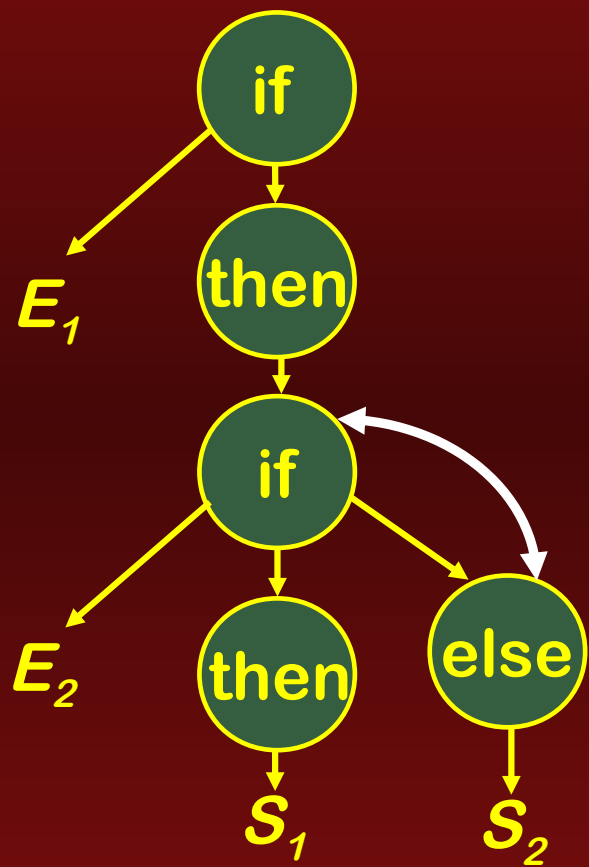
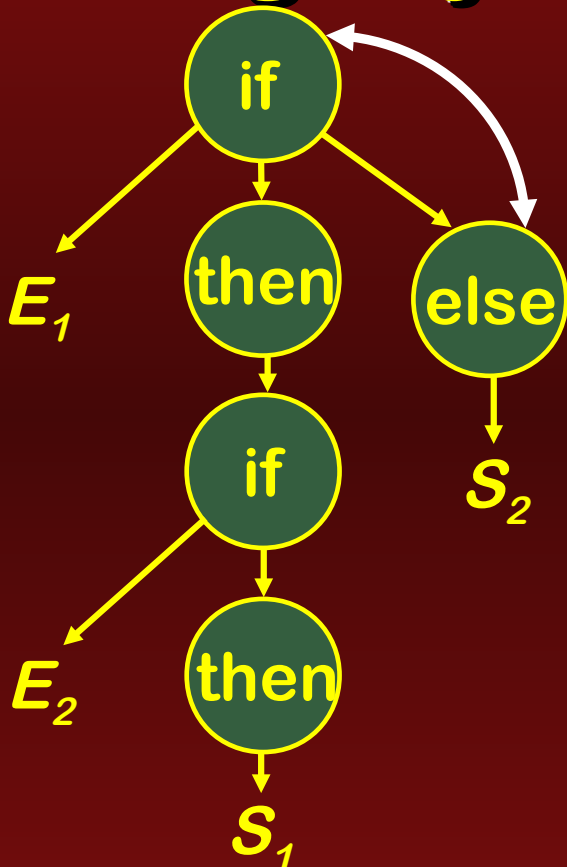
Ambiguity

Production 2,
then
Production 1:
if E_1 then
 if E_2 then S_1
 else S_2



33

Ambiguity



34

Removing Ambiguity

- Must **rewrite** grammar to avoid generating the problem
- Match each else to innermost unmatched if

35

Removing Ambiguity

- Must **rewrite** grammar to avoid generating the problem
- Match each else to innermost unmatched if

36

Removing Ambiguity

1. *Stmt* $\rightarrow$ if *E* then *Stmt*
2. | if *E* then *WithElse* else *Stmt*
3. | *Assignment*
4. *WithElse* $\rightarrow$ if *E* then *WithElse* else *WithElse*
5. | *Assignment*

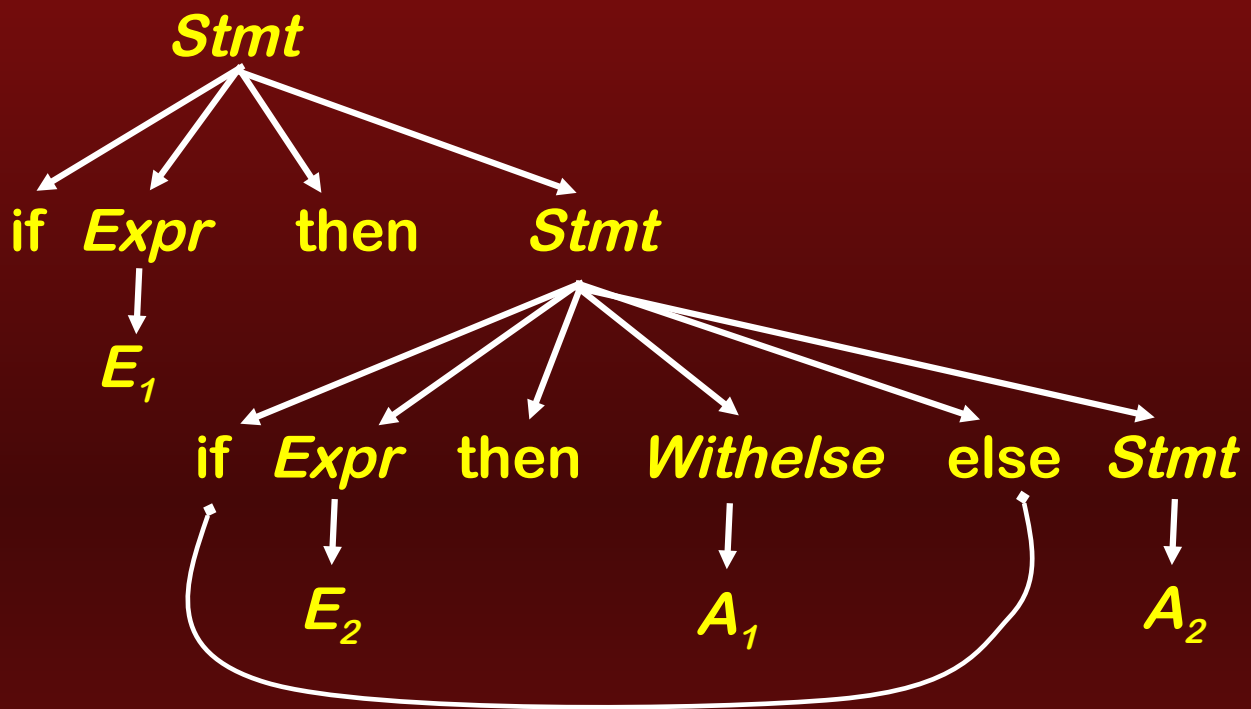
37

Removing Ambiguity

Let derive the following using the rewritten grammar:

if E_1 then if E_2 then A_1 else A_2

38



This binds the else controlling A_2 to inner if

39

Context-Free Grammars

- We have been using the term context-free without explaining why such rules are in fact “free of context”.

40

Context-Free Grammars

- The simple reason is that nonterminals appear by themselves to the left of the arrow in context-free rules:

$$A \rightarrow \alpha$$

41

Context-Free Grammars

- The rule $A \rightarrow \alpha$ says that A may be replaced by α *anywhere*, regardless of where A occurs.

42

Context-Free Grammars

- On the other hand, we could define a context as pair of strings β , γ , such that a rule would apply only if β occurs before and γ occurs after the nonterminal A .

43

Context-Sensitive Grammars

- We would write this as

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

- Such a rule in which $\alpha \neq \varepsilon$ is called a *context-sensitive* grammar rule

44

Context-Sensitive Grammars

- We would write this as

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

- Such a rule in which $\alpha \neq \varepsilon$ is called a *context-sensitive* grammar rule

45

Parsing Techniques

Parsing Techniques

Top-down parsers

- Start at the root of the parse tree and grow towards leaves.
- Pick a production and try to match the input

47

Parsing Techniques

Top-down parsers

- Start at the root of the parse tree and grow towards leaves.
- Pick a production and try to match the input

48

Parsing Techniques

Top-down parsers

- Start at the **root** of the **parse tree** and grow towards leaves.
- Pick a production and try to match the input

49

Parsing Techniques

Top-down parsers

- Bad “pick” $\Rightarrow$ may need to **backtrack**
- Some grammars are backtrack-free.

50

Parsing Techniques

Top-down parsers

- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free.

51

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

52

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

53

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

54

Parsing Techniques

Parsing Techniques

Top-down parsers

- Start at the root of the parse tree and expand to the leaves.
- Pick a production and try to match the input

Parsing Techniques

Top-down parsers

- Start at the **root** of the **parse tree** and grow towards **leaves**. Pick a production and try to match the input

3

Parsing Techniques

Top-down parsers

- Start at the **root** of the **parse tree** and grow towards **leaves**.
- Pick a production and try to match the input

4

Parsing Techniques

Top-down parsers

- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free.

5

Parsing Techniques

Top-down parsers

- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free.

6

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

7

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

8

Parsing Techniques

Bottom-up parsers

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state.

9

Compiler Construction

Sohail Aslam

Lecture 13

Parsing Techniques

Bottom-up parsers

- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

11

Parsing Techniques

Bottom-up parsers

- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

12

Top-Down Parser

- A top-down parser **starts** with the **root** of the parse tree.
- The root node is labeled with the goal symbol of the grammar

13

Top-Down Parser

- A top-down parser **starts** with the **root** of the parse tree.
- The root node is labeled with the goal symbol of the grammar

14

Top-Down Parsing Algorithm

- Construct the root node of the parse tree

parse tree matches input string

15

Top-Down Parsing Algorithm

- Construct the root node of the parse tree
- **Repeat** until the **fringe** of the parse tree matches input string

16

Top-Down Parsing

- At a **node** labeled **A**, select a production with **A** on its **lhs**
- for each symbol on its **rhs**, construct the appropriate child

17

Top-Down Parsing

- At a **node** labeled **A**, select a production with **A** on its **lhs**
- for **each** symbol on its **rhs**, **construct** the appropriate child

18

Top-Down Parsing

- When a **terminal** symbol is added to the fringe and it **does not match** the fringe, **backtrack**
- Find the next node to be expanded

19

Top-Down Parsing

- When a **terminal** symbol is added to the fringe and it **does not match** the fringe, **backtrack**
- Find the next node to be expanded

20

Top-Down Parsing

- The **key** is picking right production in step 1.

That choice should be guided by the input string

21

Top-Down Parsing

- The **key** is picking right production in step 1.
- That choice should be **guided** by the **input string**

22

Expression Grammar

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>expr</i> + <i>term</i>
3			<i>expr</i> - <i>term</i>
4			<i>term</i>
5	<i>term</i>	→	<i>term</i> * <i>factor</i>
6			<i>term</i> / <i>factor</i>
7			<i>factor</i>
8	<i>factor</i>	→	<u>number</u>
9			<u>id</u>
10			(<i>expr</i>)

23

Top-Down Parsing

- Let's try parsing

$x - 2 * y$

24

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr + term	$\uparrow \underline{x} - \underline{2} * y$
4	term + term	$\uparrow \underline{x} - \underline{2} * y$
7	factor + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> + term	$\underline{x} \uparrow - \underline{2} * y$

25

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr + term	$\uparrow \underline{x} - \underline{2} * y$
4	term + term	$\uparrow \underline{x} - \underline{2} * y$
7	factor + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> + term	$\underline{x} \uparrow - \underline{2} * y$

This worked well except that “-” does not match “+”

26

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr + term	$\uparrow \underline{x} - \underline{2} * y$
4	term + term	$\uparrow \underline{x} - \underline{2} * y$
7	factor + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> + term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> <u>+</u> term	$\underline{x} \uparrow \underline{-} \underline{2} * y$



The parser must backtrack to here

27

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr - term	$\uparrow \underline{x} - \underline{2} * y$
4	term - term	$\uparrow \underline{x} - \underline{2} * y$
7	factor - term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> - term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> <u>-</u> term	$\underline{x} \uparrow \underline{-} \underline{2} * y$



This time the “-” and “-” matched

28

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr – term	$\uparrow \underline{x} - \underline{2} * y$
4	term – term	$\uparrow \underline{x} - \underline{2} * y$
7	factor – term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> – term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> – term	$\underline{x} \uparrow - \underline{2} * y$
-	<id, <u>x</u>> – term	$\underline{x} - \uparrow \underline{2} * y$

We can advance past “**–**” to look at “**2**”

29

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	Goal	$\uparrow \underline{x} - \underline{2} * y$
1	expr	$\uparrow \underline{x} - \underline{2} * y$
2	expr – term	$\uparrow \underline{x} - \underline{2} * y$
4	term – term	$\uparrow \underline{x} - \underline{2} * y$
7	factor – term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> – term	$\uparrow \underline{x} - \underline{2} * y$
9	<id, <u>x</u>> – term	$\underline{x} \uparrow - \underline{2} * y$
-	<id, <u>x</u>> – term	$\underline{x} - \uparrow \underline{2} * y$

Now, we need to expand “**term**”

30

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	$\langle \text{id}, \underline{x} \rangle$ – <i>term</i>	$\underline{x} - \uparrow \underline{2} * y$
7	$\langle \text{id}, \underline{x} \rangle$ – <i>factor</i>	$\underline{x} - \uparrow \underline{2} * y$
9	$\langle \text{id}, \underline{x} \rangle$ – $\langle \text{num}, \underline{2} \rangle$	$\underline{x} - \uparrow \underline{2} * y$
-	$\langle \text{id}, \underline{x} \rangle$ – $\langle \text{num}, 2 \rangle$	$\underline{x} - \underline{2} \uparrow * y$

We have more input but no non-terminals left to expand

31

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	$\langle \text{id}, \underline{x} \rangle$ – <i>term</i>	$\underline{x} - \uparrow \underline{2} * y$
7	$\langle \text{id}, \underline{x} \rangle$ – <i>factor</i>	$\underline{x} - \uparrow \underline{2} * y$
9	$\langle \text{id}, \underline{x} \rangle$ – $\langle \text{num}, \underline{2} \rangle$	$\underline{x} - \uparrow \underline{2} * y$
-	$\langle \text{id}, \underline{x} \rangle$ – $\langle \text{num}, 2 \rangle$	$\underline{x} - \underline{2} \uparrow * y$

- The expansion terminated too soon
- $\Rightarrow$ Need to backtrack

32

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	$\langle \text{id}, \underline{x} \rangle - \text{term}$	$\underline{x} - \uparrow \underline{2} * y$
◆ 5	$\langle \text{id}, \underline{x} \rangle - \text{term} * \text{factor}$	$\underline{x} - \uparrow \underline{2} * y$
7	$\langle \text{id}, \underline{x} \rangle - \text{factor} * \text{factor}$	$\underline{x} - \uparrow \underline{2} * y$
8	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, 2 \rangle * \text{factor}$	$\underline{x} - \uparrow \underline{2} * y$
-	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, 2 \rangle * \text{factor}$	$\underline{x} - \underline{2} \uparrow * y$
-	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, 2 \rangle * \text{factor}$	$\underline{x} - \underline{2} * \uparrow y$
9	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$	$\underline{x} - \underline{2} * \uparrow y$
-	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$	$\underline{x} - \underline{2} * y \uparrow$

Success! We matched and consumed all the input

33

Another Possible Parse

<i>P</i>	<i>Sentential Form</i>	<i>input</i>
-	<i>Goal</i>	$\uparrow \underline{x} - \underline{2} * y$
1	<i>expr</i>	$\uparrow \underline{x} - \underline{2} * y$
2	$\text{expr} + \text{term}$	$\uparrow \underline{x} - \underline{2} * y$
2	$\text{expr} + \text{term} + \text{term}$	$\uparrow \underline{x} - \underline{2} * y$
2	$\text{expr} + \text{term} + \text{term} + \text{term}$	$\uparrow \underline{x} - \underline{2} * y$
2	$\text{expr} + \text{term} + \text{term} + \text{term} + \dots$	$\uparrow \underline{x} - \underline{2} * y$

Wrong! Parser must make the right choice

34

Left Recursion

Top-down parsers cannot
handle left-recursive
grammars

35

Left Recursion

Formally,

A grammar is **left recursive**
if $\exists A \in NT$ such that $\exists$ a
derivation $A \Rightarrow^* A \alpha$, for
some string $\alpha \in (NT \cup T)^*$

36

Left Recursion

- Our expression grammar is left recursive.

termination in a top-down parser

37

Left Recursion

- Our expression grammar is left recursive.
- This can lead to non-termination in a top-down parser

38

Left Recursion

- Non-termination is bad in any part of a compiler!

39

Left Recursion

- For a top-down parser, any recursion must be a *right recursion*
- We would like to convert left recursion to right recursion

40

Left Recursion

- For a top-down parser, any recursion must be a *right recursion*
- We would like to convert left recursion to right recursion

41

Eliminating Left Recursion

To remove left recursion, we **transform** the grammar

42

Eliminating Left Recursion

Consider a grammar fragment:

$$\begin{array}{c} A \rightarrow A \alpha \\ | \beta \end{array}$$

where neither α nor β starts with A .

43

Eliminating Left Recursion

We can rewrite this as:

$$A \rightarrow \beta A'$$

$$\begin{array}{c} A' \rightarrow \alpha A' \\ | \varepsilon \end{array}$$

where A' is a new non-terminal

44

Eliminating Left Recursion

We can rewrite this as:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

where A' is a new non-terminal

45

Eliminating Left Recursion

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

- This accepts the same language but uses only right recursion

46

Eliminating Left Recursion

The expression grammar we have been using contains two cases of left- recursion

47

Eliminating Left Recursion

<i>expr</i>	→	<i>expr</i> + <i>term</i>
		<i>expr</i> – <i>term</i>
		<i>term</i>

<i>term</i>	→	<i>term</i> * <i>factor</i>
		<i>term</i> / <i>factor</i>
		<i>factor</i>

48

Eliminating Left Recursion

Applying the transformation yields

$$\begin{aligned} \text{◆ } \textit{expr} &\rightarrow \textit{term expr}' \\ \textit{expr}' &\rightarrow \begin{array}{l} + \textit{term expr}' \\ - \textit{term expr}' \\ \varepsilon \end{array} \end{aligned}$$

49

Eliminating Left Recursion

Applying the transformation yields

$$\begin{aligned} \textit{term} &\rightarrow \textit{factor term}' \\ \textit{term}' &\rightarrow \begin{array}{l} * \textit{factor term}' \\ / \textit{factor term}' \\ \varepsilon \end{array} \end{aligned}$$

50

Eliminating Left Recursion

- These fragments use only **right recursion**

▪ They retain the original **left associativity**

- A top-down parser will terminate using them.

51

Eliminating Left Recursion

- These fragments use only **right recursion**

- They **retain** the original **left associativity**

- A top-down parser will terminate using them.

52

Eliminating Left Recursion

- These fragments use only **right recursion**
- They **retain** the original **left associativity**
- A top-down **parser** will **terminate** using them.

53

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>– term expr'</i>
5			<i>ε</i>
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			<i>ε</i>
10	<i>factor</i>	→	<u><i>number</i></u>
11			<u><i>id</i></u>
12			<i>(expr)</i>

54

Predictive Parsing

- If a top down parser picks the **wrong production**, it may need to **backtrack**
- Alternative is to *look ahead* in input and use context to pick correctly

55

Predictive Parsing

- If a top down parser picks the **wrong production**, it may need to **backtrack**
- Alternative is to *look ahead* in **input** and use context to pick correctly

56

Predictive Parsing

- How much lookahead is needed?

large amount

57

Predictive Parsing

- How much lookahead is needed?
- In general, an arbitrarily large amount

58

Predictive Parsing

- Fortunately, large classes of CFGs can be parsed with **limited lookahead**
- Most programming languages constructs fall in those subclasses

59

Predictive Parsing

- Fortunately, large classes of CFGs can be parsed with **limited lookahead**
- Most **programming languages** constructs fall in those subclasses

60

Predictive Parsing

Basic Idea:

Given $A \rightarrow \alpha \mid \beta$,
the parser should be
able to choose
between α and β .

61

Predictive Parsing

FIRST Sets:

For some rhs $\alpha \in G$, define
FIRST(α) as the set of
tokens that appear as the
first symbol in some string
that derives from α .

62

Predictive Parsing

That is,

$$\underline{x} \in \text{FIRST}(\alpha)$$

iff $\alpha \Rightarrow^* \underline{x} \gamma$, for some γ .

63

Compiler Construction

Sohail Aslam

Lecture 14

Predictive Parsing

The LL(1) Property

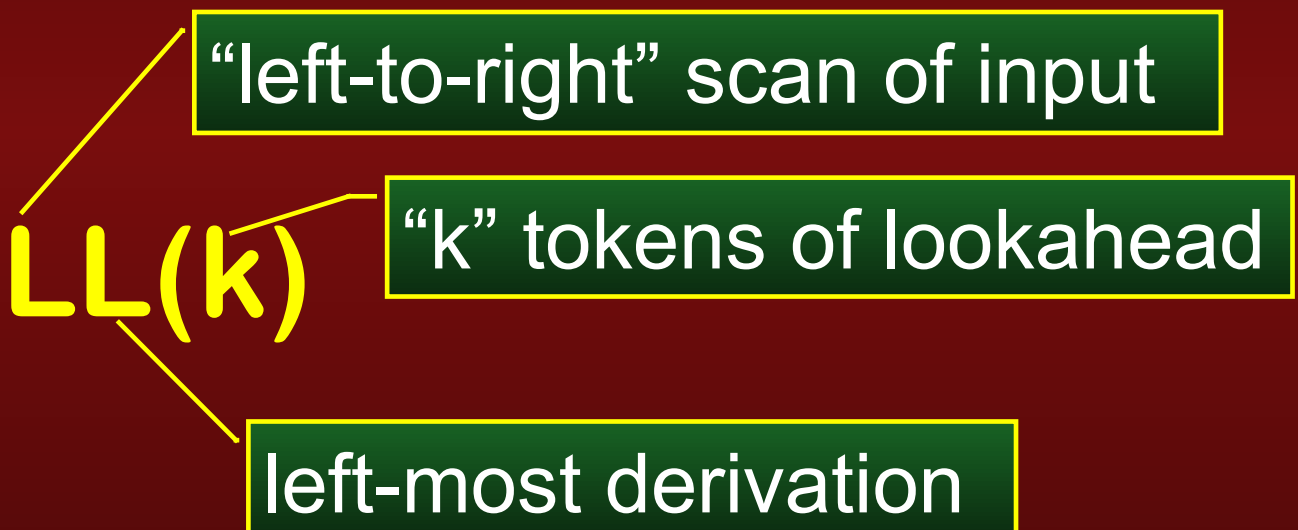
If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

2

Predictive Parsing

Predictive parsers accept LL(k) grammars



3

Predictive Parsing

The LL(1) Property

$$\mathbf{FIRST}(\alpha) \cap \mathbf{FIRST}(\beta) = \emptyset$$

allows the parser to make a
correct choice with a
lookahead of exactly *one*
symbol!

4

Predictive Parsing

What about ϵ -productions?

$\Rightarrow$ They complicate the
implementation of LL(1)

5

Predictive Parsing

What about ε -productions?

$\Rightarrow$ They complicate the definition of **LL(1)**

6

Predictive Parsing

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in \text{FIRST}(\alpha)$, then we need to ensure that **FIRST(β)** is disjoint from **FOLLOW(α)**, too

7

Predictive Parsing

FOLLOW(α)

is the set of all words in the grammar that can legally appear **after** an α .

8

Predictive Parsing

For a non-terminal X ,

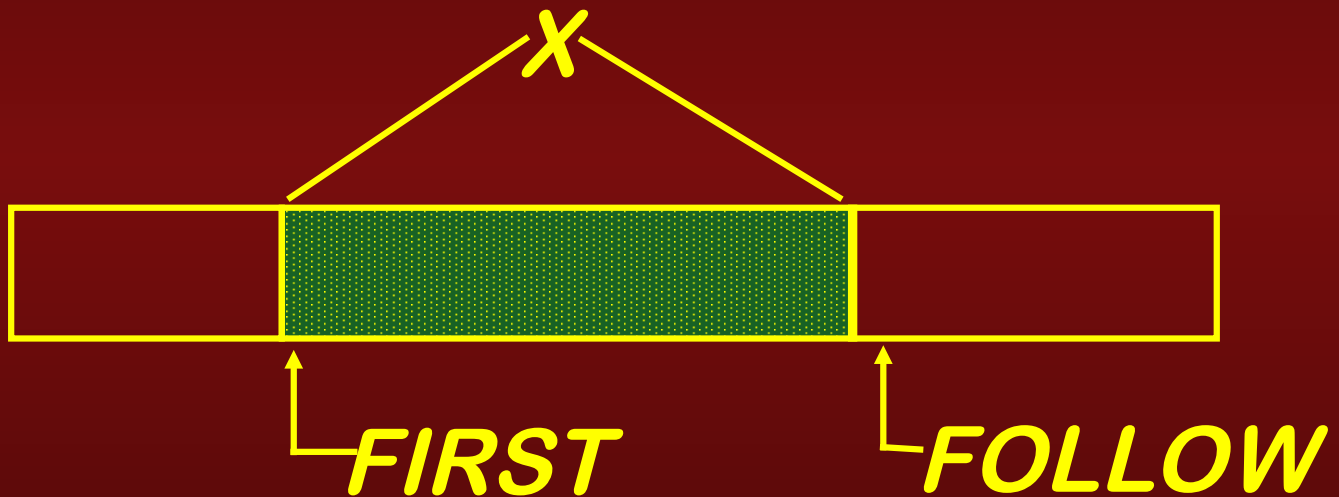
FOLLOW(X)

is the set of symbols that might **follow** the derivation of X .

9

Predictive Parsing

FIRST and FOLLOW



10

Predictive Parsing

Define $\text{FIRST}^+(\alpha)$ as

$\text{FIRST}(\alpha) \cup \text{FOLLOW}(\alpha)$, if

$\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$, otherwise

11

Predictive Parsing

Then a grammar is **LL(1)**
iff **$A \rightarrow \alpha$** and **$A \rightarrow \beta$**
implies

$$\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset$$

12

Predictive Parsing

Given a grammar that has
the **LL(1)** property

- we can write a parsing routine to recognize each lhs
- code is simple and fast

13

Predictive Parsing

Given a grammar that has the is **LL(1)** property

- we can write a simple routine to recognize each lhs
- code is simple and fast

14

Predictive Parsing

Given a grammar that has the is **LL(1)** property

- we can write a simple routine to recognize each lhs
- code is simple and fast

15

Predictive Parsing

Consider

$$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3,$$

which satisfies the LL(1)

property $\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset$

16

/* find an A */

if(token $\in \text{FIRST}(\beta_1)$)

 find a β_1 and return true

else if(token $\in \text{FIRST}(\beta_2)$)

 find a β_2 and return true

if(token $\in \text{FIRST}(\beta_3)$)

 find a β_3 and return true

else error and return false

17

/* find an **A** */

if(**token** $\in$ **FIRST(β_1)**)

find a β_1 and return true

else if(token $\in$ FIRST(β_2))

find a β_2 and return true

if(token $\in$ FIRST(β_3))

find a β_3 and return true

else error and return false

18

/* find an **A** */

if(**token** $\in$ **FIRST(β_1)**)

find a β_1 and return true

else if(**token** $\in$ **FIRST(β_2)**)

find a β_2 and return true

if(token $\in$ FIRST(β_3))

find a β_3 and return true

else error and return false

19

/* find an **A** */

if(**token** $\in$ **FIRST(β_1)**)

find a β_1 and return true

else if(**token** $\in$ **FIRST(β_2)**)

find a β_2 and return true

if(**token** $\in$ **FIRST(β_3)**)

find a β_3 and return true

else error and return false

20

/* find an **A** */

if(**token** $\in$ **FIRST(β_1)**)

find a β_1 and return true

else if(**token** $\in$ **FIRST(β_2)**)

find a β_2 and return true

if(**token** $\in$ **FIRST(β_3)**)

find a β_3 and return true

else error and return false

21

Predictive Parsing

Grammar with the **LL(1)** property are called *predictive grammars* because the parser can “**predict**” the correct expansion at each point in the parse.

22

Predictive Parsing

- Parsers that capitalize on the **LL(1)** property are called *predictive parsers*
- One kind of predictive parser is the *recursive descent* parser

23

Predictive Parsing

- Parsers that capitalize on the **LL(1)** property are called *predictive parsers*
- One kind of predictive parser is the *recursive descent* parser

24

Recursive Descent Parsing

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			ϵ
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			ϵ
10	<i>factor</i>	→	<u>number</u>
11			<u>id</u>
12			<i>(expr)</i>

25

Recursive Descent Parsing

This leads to a parser with six mutually recursive routines

<i>Goal</i>	<i>Term</i>
<i>Expr</i>	<i>TPrime</i>
<i>EPrime</i>	<i>Factor</i>

26

Recursive Descent Parsing

Each recognizes one non-terminal (NT) or terminal (T)

<i>Goal</i>	<i>Term</i>
<i>Expr</i>	<i>TPrime</i>
<i>EPrime</i>	<i>Factor</i>

27

Recursive Descent Parsing

- The term *descent* refers to the direction in which the *parse tree* is built.
- Here are some of these routines written as functions

28

Recursive Descent Parsing

- The term *descent* refers to the direction in which the *parse tree* is built.
- Here are some of these routines written as functions

29

```
Goal() {  
    token = next_token();  
    if(Expr() == true && token == EOF)  
        next compilation step  
    else {  
        report syntax error;  
        return false;  
    }  
}
```

30

```
Expr()  
{  
    if(Term() == false)  
        return false;  
    else  
        return Eprime();  
}
```

31

```
Eprime() {  
    token_type op = next_token();  
    if( op == PLUS || op == MINUS ) {  
        if(Term() == false)  
            return false;  
        else  
            return Eprime();  
    }  
}
```

32

Recursive Descent Parsing

Functions for other non-terminals **Term**, **Factor**, **Tprime** follow the same pattern.

33

Recursive Descent in C++

Shortcomings

- Too procedural
- No convenient way to build parse tree

34

Recursive Descent in C++

Using an OO Language

- Associate a class with each non-terminal
- Allocated object contains pointer to the parse tree

35

Recursive Descent in C++

Using an OO Language

- Associate a **class** with **each non-terminal** symbol
- Allocated object contains pointer to the parse tree

36

Recursive Descent in C++

Using an OO Language

- Associate a **class** with **each non-terminal** symbol
- Allocated object contains pointer to the **parse tree**

37

Non-terminal Classes

```
class NonTerminal {
public:
    NonTerminal(Scanner* sc) {
        s = sc; tree = NULL; }
    virtual ~NonTerminal() {}
    virtual bool isPresent()=0;
    TreeNode* AST() {
        return tree;
    }
}
```

38

```
class NonTerminal {
public:
```

```
    NonTerminal(Scanner* sc) {
        s = sc; tree = NULL; }
```

Constructor: stores the pointer to the scanner (lexer) in protected variable and initializes tree pointer to NULL.

39

```
class NonTerminal {
public:
    NonTerminal(Scanner* sc) {
        s = sc; tree = NULL; }
    virtual ~NonTerminal() {}
};
```

*Polymorphic default destructor.
Called in case derived classes
do not define their own*

40

```
class NonTerminal {
public:
```

***isPresent** is pure virtual (=0).
Base class will not provide
implementation*

```
virtual bool isPresent()=0;
    TreeNode* AST() {
        return tree;
    }
};
```

41


```
class NonTerminal {  
public:
```

*Return pointer to Abstract
Syntax tree. Available to all
subclasses*

```
TreeNode* AST() {  
    return tree;  
}
```

42

```
class NonTerminal {  
public:  
    NonTerminal(Scanner* sc) {  
        s = sc; tree = NULL; }  
    virtual ~NonTerminal() {}  
    virtual bool isPresent()=0;  
    TreeNode* AST() {  
        return tree;  
    }  
}
```

43

```
class NonTerminal {  
    . . . . .  
    . . . . .  
protected:  
    Scanner* s;  
    TreeNode* tree;  
}
```

44

```
class Expr:public NonTerminal  
{  
public:  
    Expr(Scanner* sc):  
        NonTerminal(sc){ }  
    virtual bool isPresent();  
}
```

45

```
class Expr:public NonTerminal
{
```

NonTerminal is the base class; Expr is the derived class.

```
}
```

46

```
class Expr:public NonTerminal
{
public:
```

```
    Expr(Scanner* sc) :
        NonTerminal(sc) { }
```

Expr's constructor calls the base class constructor explicitly.

47

```
class Expr:public NonTerminal
{
```

isPresent() is a
polymorphic function.

```
NonTerminal(sc) { }
```

```
virtual bool isPresent();
```

```
}
```

48

```
class Eprime:public
                NonTerminal {
```

```
public:
```

```
    Eprime(Scanner* sc,
            TreeNode* t):
```

```
        NonTerminal(sc) {
            exprSofar = t; }
```

```
    virtual bool isPresent();
```

```
protected:
```

```
    TreeNode* exprSofar;
```

```
}
```

49

```
class Term:public NonTerminal
{
public:
    Term(Scanner* sc):
        NonTerminal(sc){ }
    virtual bool isPresent();
}
```

50

```
class Tprime:public
                    NonTerminal {
public:
    Tprime(Scanner* sc,
           TreeNode* t):
        NonTerminal(sc){
            exprSofar = t; }
    virtual bool isPresent();
protected:
    TreeNode* exprSofar;
}
```

51

```
class Factor:public  
    NonTerminal {  
public:  
    Factor(Scanner* sc,  
           TreeNode* t):  
        NonTerminal(sc){ };  
  
    virtual bool isPresent();  
}
```

52

Compiler Construction

Sohail Aslam

Lecture 15

Recursive Descent Parser

- Let's consider the implementation of the C++ classes for the non-terminal
- Start with **Expr**; recall the grammar

2

Recursive Descent Parser

- Let's consider the implementation of the C++ classes for the non-terminal
- Start with **Expr**; recall the grammar

3

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			<i>ε</i>
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			<i>ε</i>
10	<i>factor</i>	→	<u><i>number</i></u>
11			<u><i>id</i></u>
12			<i>(expr)</i>

4

```

bool Expr::isPresent() {
    Term* op1 = new Term(s);
    if(!op1->isPresent())
        return false;
    tree = op1->AST();
    Eprime* op2 =
        new Eprime(s, tree);
    if(op2->isPresent())
        tree = op2->AST();
    return true;
}

```

5

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			<i>ε</i>
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			<i>ε</i>
10	<i>factor</i>	→	<u><i>number</i></u>
11			<u><i>id</i></u>
12			<i>(expr)</i>

6

```

bool Eprime::isPresent()
{
    int op=s->nextToken();
    if(op==PLUS || op==MINUS) {
        s->advance();
        Term* op2=new Term(s);
        if(!op2->isPresent())
            syntaxError(s);
        TreeNode* t2=op2->AST();
        tree = new
            TreeNode(op,exprSofar,t2);
        . . . . .
    }
}

```

7

.

```
Eprime* op3 =  
    new Eprime(s, tree);  
    if(op3->isPresent())  
        tree = op3->AST();  
    return true;  
}  
else return false;  
}
```

8

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			<i>ε</i>
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			<i>ε</i>
10	<i>factor</i>	→	<u>number</u>
11			<u>id</u>
12			<i>(expr)</i>

9

```

bool Term::isPresent()
{
    Factor* op1 = new Factor(s);
    if(!op1->isPresent())
        return false;
    tree = op1->AST();
    Tprime* op2 =
        new Tprime(s, tree);
    if(op2->isPresent())
        tree = op2->AST();
    return true;
}

```

10

```

bool Tprime::isPresent()
{
    int op=s->nextToken();
    if(op == MUL || op == DIV) {
        s->advance();
        Factor* op2=new Factor(s);
        if(!op2->isPresent())
            syntaxError(s);
        TreeNode* t2=op2->AST();
        tree = new
            TreeNode(op,exprSofar,t2);
        .....
    }
}

```

11

• •

```
Tprime* op3 =  
    new Tprime(s, tree);  
    if(op3->isPresent())  
        tree = op3->AST();  
    return true;  
}  
else return false;  
}
```

12

1	<i>Goal</i>	→	<i>expr</i>
2	<i>expr</i>	→	<i>term expr'</i>
3	<i>expr'</i>	→	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			ε
6	<i>term</i>	→	<i>factor term'</i>
7	<i>term'</i>	→	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			ε
10	<i>factor</i>	→	<u>number</u>
11			<u>id</u>
12			(<i>expr</i>)

13

```

bool Factor::isPresent()
{
    int op=s->nextToken();
    if(op == ID || op == NUM)
    {
        tree = new
        TreeNode(op,s->tokenValue());
        s->advance();
        return true;
    }
    . . . . .

```

14

```

if( op == LPAREN ) {
    s->advance();
    Expr* opr = new Expr(s);
    if(!opr->isPresent())
        syntaxError(s);
    if(s->nextToken() != RPAREN)
        syntaxError(s);
    s->advance();
    tree = opr->AST();
    return true;
}
return false;
}

```

15

Compiler Construction

Sohail Aslam

Lecture 16

Example: $2+4*6$

>> Expr::isPresent()

>> Term::isPresent()

>> Factor::isPresent()

token: 2 (257)

<< Factor::isPresent() return true

>> Tprime::isPresent()

token: + (267)

<< Tprime::isPresent() return false

<< Term::isPresent() return true

Example: $2+4*6$

```
>> Eprime::isPresent() ◆  
    token: + (267)  
    >> Term::isPresent()  
        >> Factor::isPresent()  
            token: 4 (257)  
        << Factor::isPresent() return true  
    >> Tprime::isPresent()  
        token: * (269)  
        >> Factor::isPresent()  
            token: 6 (257)  
        << Factor::isPresent() return true
```

3

Example: $2+4*6$

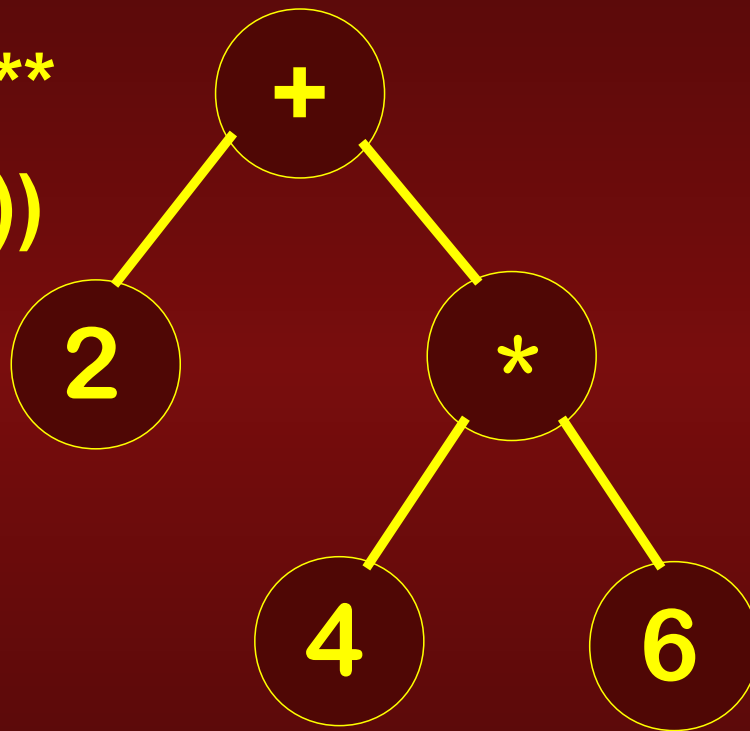
```
    >> Tprime::isPresent() ◆  
        token: (0)  
    << Tprime::isPresent() return false  
    << Tprime::isPresent() return true  
    << Term::isPresent() return true  
    >> Eprime::isPresent()  
        token: (0)  
    << Eprime::isPresent() return false  
    << Eprime::isPresent() return true  
    << Expr::isPresent() return true
```

4

Example: $2+4*6$

**** AST ****

$(2+(4*6))$



5

Nonrecursive Predictive Parsing

- It is possible to build a **nonrecursive** predictive parser
- This is done by maintaining an explicit stack

1

Nonrecursive Predictive Parsing

- It is possible to build a **nonrecursive** predictive parser
- This is done by maintaining an **explicit stack**

2

Table-driven Parsers

- The nonrecursive LL(1) parser looks up the production to apply by looking up a **parsing table**

3

Table-driven Parsers

LL(1) table:

- One dimension for current non-terminal to expand
- One dimension for next token
- Table entry contains one production

4

Table-driven Parsers

LL(1) table:

- One dimension for current **non-terminal** to expand
- One dimension for **next token**
- Table entry contains one production

5

Table-driven Parsers

LL(1) table:

- One dimension for current **non-terminal** to expand
- One dimension for **next token**
- Table entry contains one **production**

6

Consider the expression grammar

1	E	$\rightarrow$	$T E'$
2	E'	$\rightarrow$	$+ T E'$
3		$ $	ε
4	T	$\rightarrow$	$F T'$
5	T'	$\rightarrow$	$* F T'$
6		$ $	ε
7	F	$\rightarrow$	(E)
8		$ $	<u>id</u>

7

Predictive Parsing Table

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \epsilon$	$T \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<i>F</i>	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		

Columns for next token

8

Predictive Parsing Table

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$ ◆			$E \rightarrow TE'$ ◆		
<i>E'</i>		$E' \rightarrow$ ◆ $+TE'$			$E' \rightarrow \epsilon$ ◆	$E' \rightarrow \epsilon$ ◆
<i>T</i>	$T \rightarrow FT'$ ◆			$T \rightarrow FT'$ ◆		
<i>T'</i>		$T' \rightarrow$ ◆ ϵ	$T \rightarrow *$ ◆ FT'		$T' \rightarrow$ ◆ ϵ	$T' \rightarrow$ ◆ ϵ
<i>F</i>	$F \rightarrow \underline{\text{id}}$ ◆			$F \rightarrow ($ ◆ $E)$		

Blank entries are errors

9

Predictive Parsers

- The predictive parser uses an **explicit stack** to keep track of pending non-terminals
- It can thus be implemented without recursion.

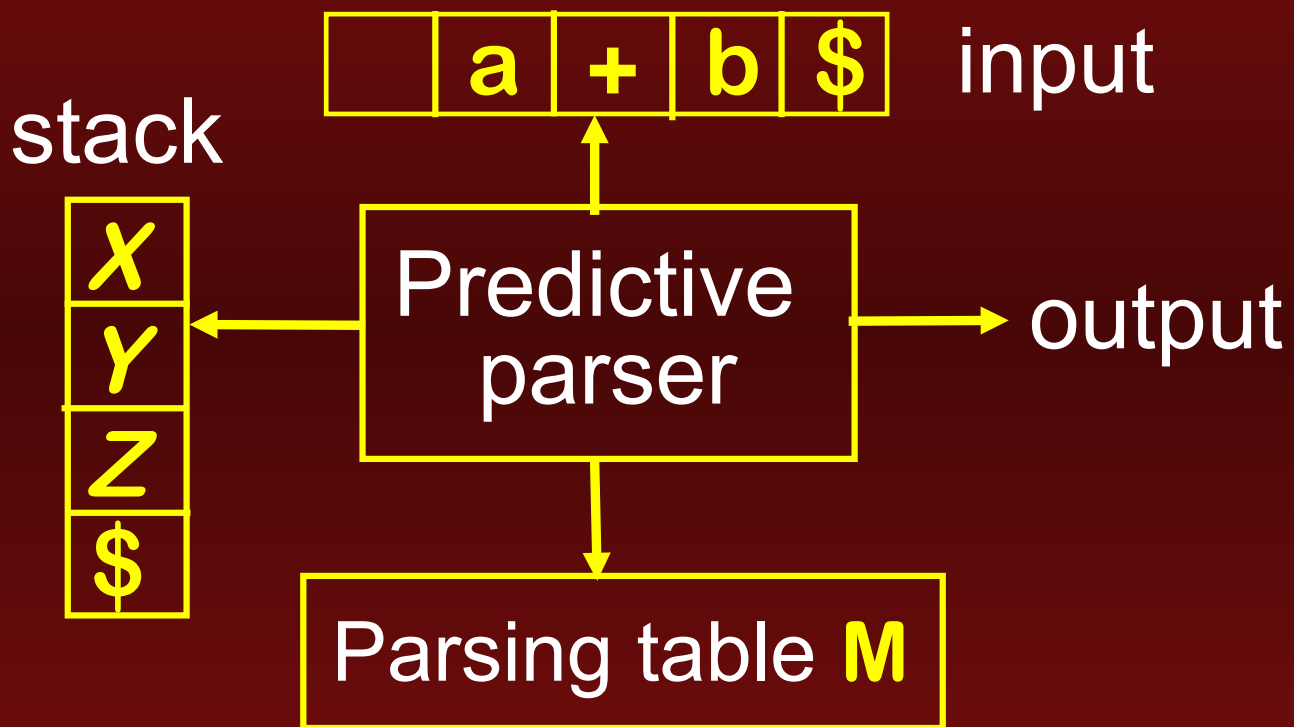
10

Predictive Parsers

- The predictive parser uses an **explicit stack** to keep track of pending non-terminals
- It can thus be implemented **without recursion.**

11

Predictive Parsers



12

LL(1) Parsing Algorithm

- The *input buffer* contains the string to be parsed; **\$** is the end-of-input marker
- The *stack* contains a sequence of grammar symbols

13

LL(1) Parsing Algorithm

- Initially, the stack contains the start symbol of the grammar on the top of \$.

14

LL(1) Parsing Algorithm

The parser is controlled by a program that behaves as follows:

15

LL(1) Parsing Algorithm

- The program considers X , the symbol on **top** of the **stack**, and **a**, the current **input** symbol.

16

LL(1) Parsing Algorithm

- These two symbols, X and **a** **determine** the **action** of the parser.
- There are *three* possibilities.

17

LL(1) Parsing Algorithm

- These two symbols, ***X*** and ***a*** determine the action of the parser.
- There are *three* possibilities.

18

LL(1) Parsing Algorithm

1. ***X = a = \$***,
the parser **halts** and
announces successful
completion.

19

LL(1) Parsing Algorithm

2. $X = a \neq \$$

the parser pops X off the stack and advances input pointer to next input symbol

20

LL(1) Parsing Algorithm

3. If X is a nonterminal, the program consults entry $M[X, a]$ of parsing table M .

21

LL(1) Parsing Algorithm

If the entry is a production

$$M[X,a] = \{X \rightarrow UVW\}$$

the parser replaces X on top of the stack by WVU (with U on top).

22

LL(1) Parsing Algorithm

As output, the parser just prints the production used:

$$X \rightarrow UVW$$

However, any other code could be executed here.

23

LL(1) Parsing Algorithm

If $M[X,a] = \text{error}$, the parser calls an error recovery routine

24

LL(1) Parsing Algorithm

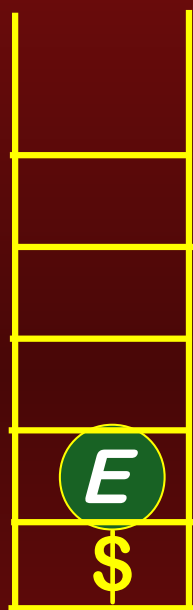
Example:

Let's parse the input string

id+id*id

using the nonrecursive
LL(1) parser

25



stack

id + id * id \$

Parsing
Table **M**

26

Predictive Parsing Table

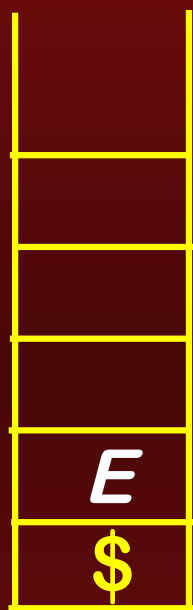
	id	+	*	(	)	\$
E	$E \rightarrow TE'$ ♦			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		

27

Compiler Construction

Sohail Aslam

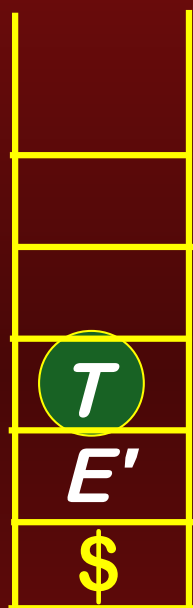
Lecture 17



stack

id + id * id \$

$E \rightarrow TE'$
Parsing
Table **M**

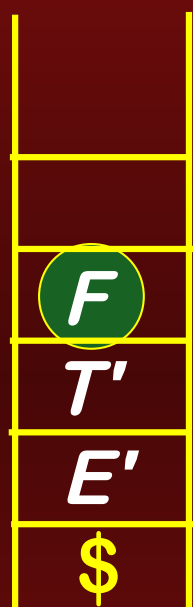


stack

$\text{id} + \text{id} * \text{id} \$$

Parsing
Table M

30

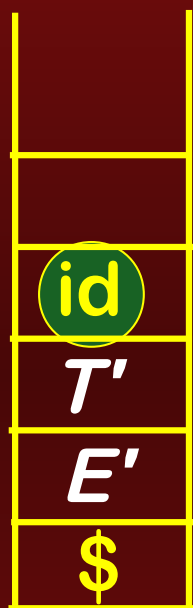


stack

$\text{id} + \text{id} * \text{id} \$$

Parsing
Table M

31

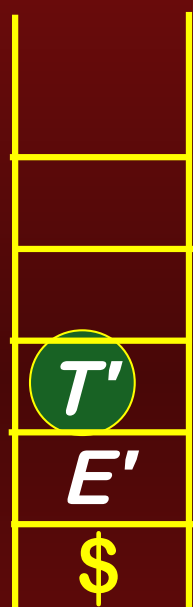


stack

id + id * id \$

Parsing
Table **M**

32

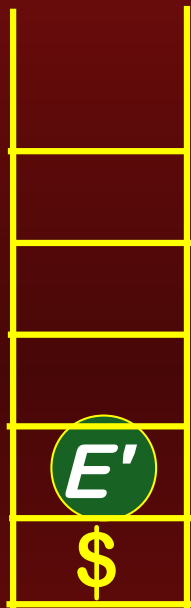


stack

+ id * id \$

$T' \xrightarrow{\epsilon}$ Parsing
Table **M**

33

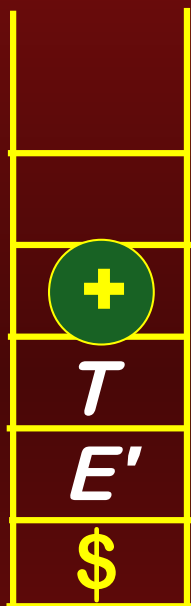


stack

$\odot id * id \$$

~~$E' \rightarrow TgE'$~~
Parsing
Table **M**

34



stack

$\odot id * id \$$

Parsing
Table **M**

35

<i>Stack</i>	<i>Input</i>	<i>Ouput</i>
$\$E$	$\underline{id}+id*id\$$	
$\$E'T$	$\underline{id}+id*id\$$	$E \rightarrow TE'$
$\$E'T'F$	$\underline{id}+id*id\$$	$T \rightarrow FT'$
$\$E'T' \underline{id}$	$\underline{id}+id*id\$$	$F \rightarrow \underline{id}$
$\$E'T'$	$+id*id\$$	
$\$E'$	$+id*id\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+id*id\$$	$E' \rightarrow +TE'$

36

<i>Stack</i>	<i>Input</i>	<i>Ouput</i>
$\$E'T$	$\underline{id}*id\$$	
$\$E'T'F$	$\underline{id}*id\$$	$T \rightarrow FT'$
$\$E'T' \underline{id}$	$\underline{id}*id\$$	$F \rightarrow \underline{id}$
$\$E'T'$	$*id\$$	
$\$E'T'F*$	$*id\$$	$T \rightarrow *FT'$
$\$E'T'F$	$\underline{id}\$$	
$\$E'T' \underline{id}$	$\underline{id}\$$	$F \rightarrow \underline{id}$

37

<i>Stack</i>	<i>Input</i>	<i>Ouput</i>
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \varepsilon$
$\$$	$\$$	$E' \rightarrow \varepsilon$

38

LL(1) Parsing Algorithm

- Note that productions output are tracing out a *leftmost derivation*
- The grammar symbols on the stack make up *left-sentential forms*

39

LL(1) Parsing Algorithm

- Note that productions output are tracing out a *leftmost derivation*
- The grammar symbols on the stack make up *left-sentential forms*

40

LL(1) Table Construction

- Top-down parsing expands a parse tree from the start symbol to the leaves
- Always expand the leftmost non-terminal

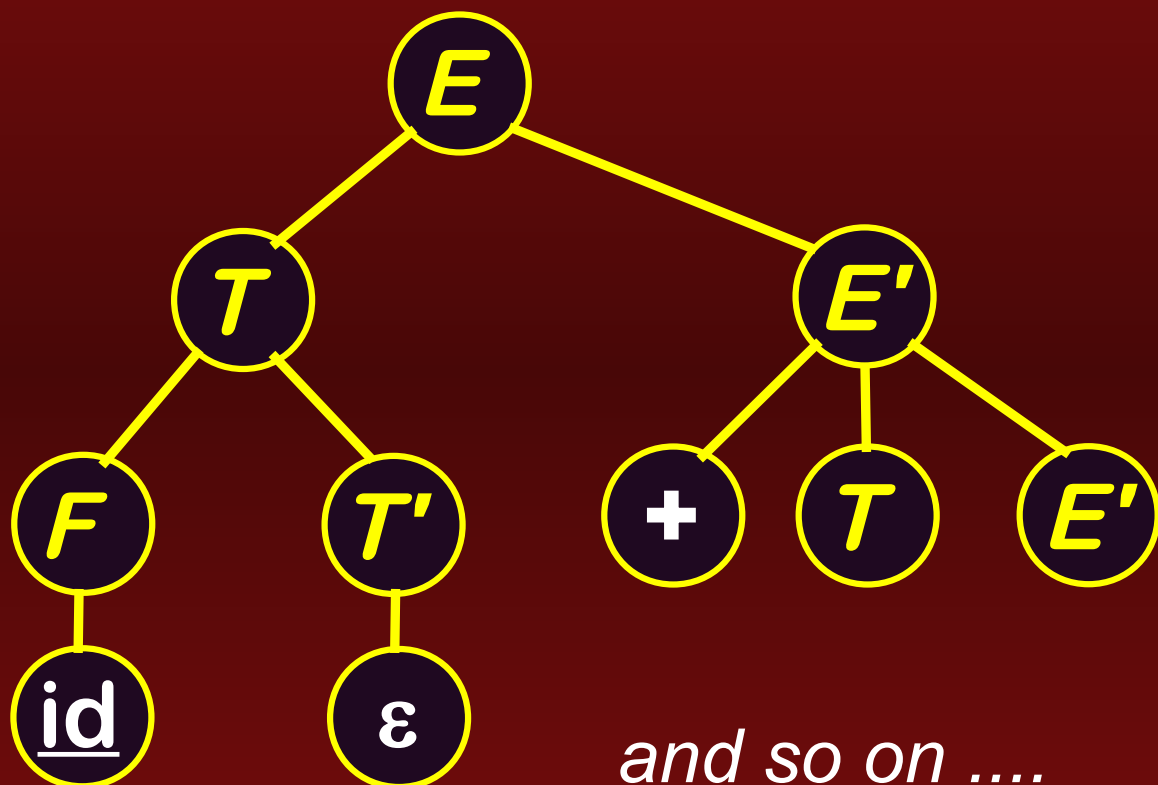
41

LL(1) Table Construction

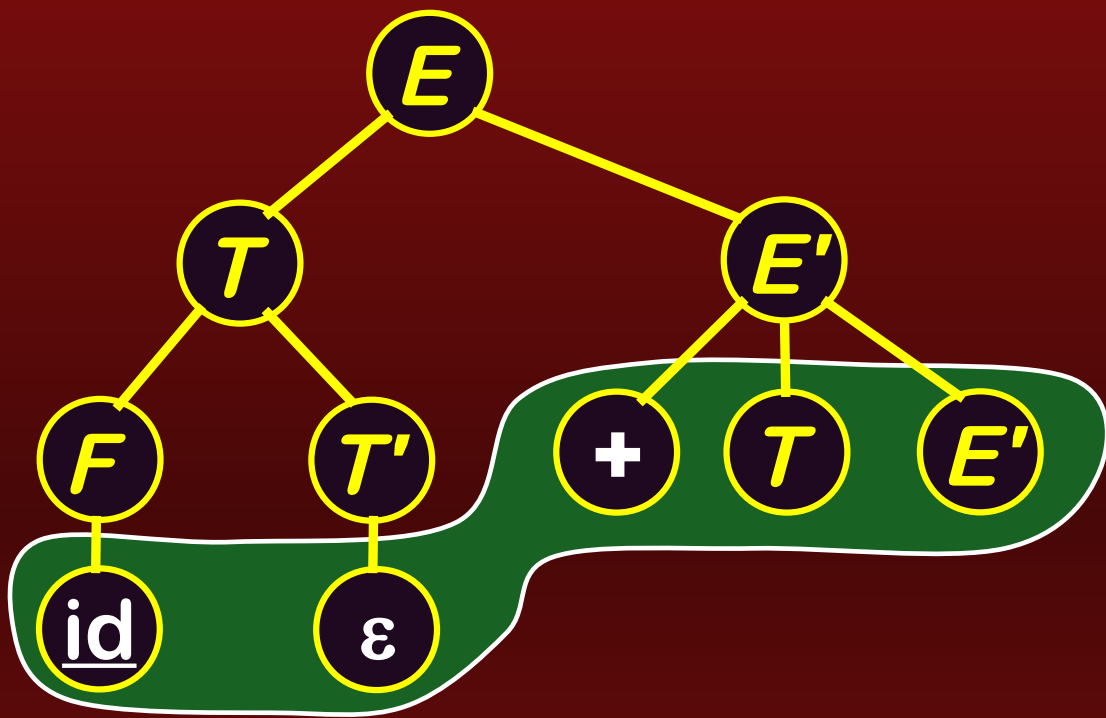
- Top-down parsing expands a parse tree from the start symbol to the leaves
- Always expand the leftmost non-terminal

42

id+id*id

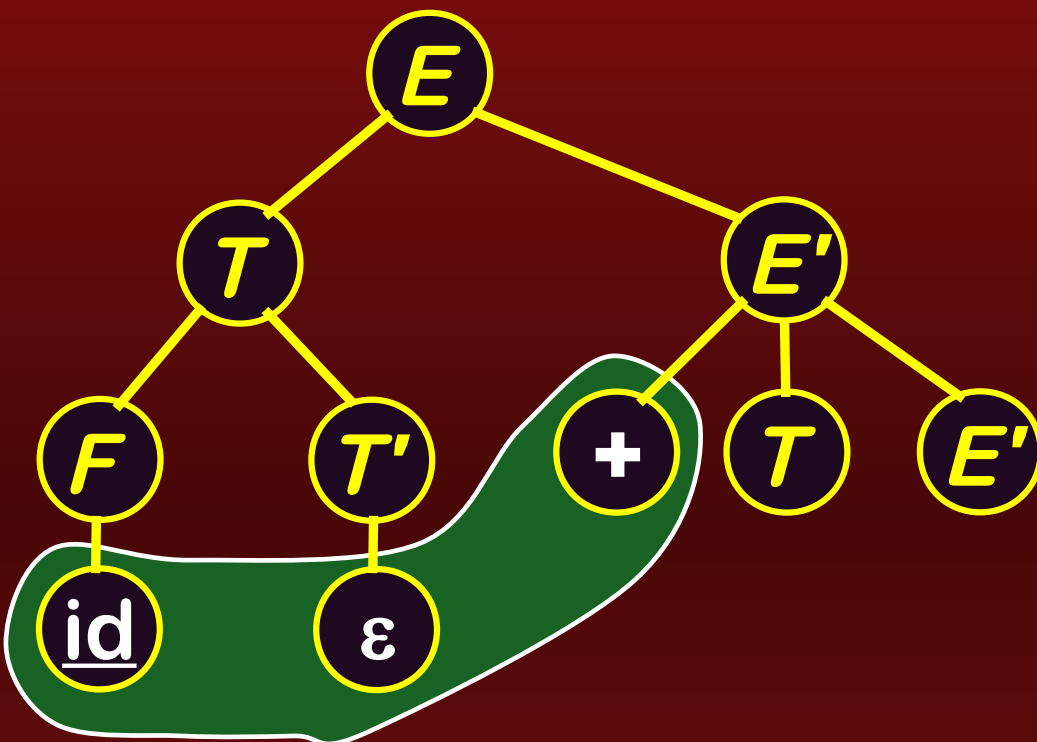


43



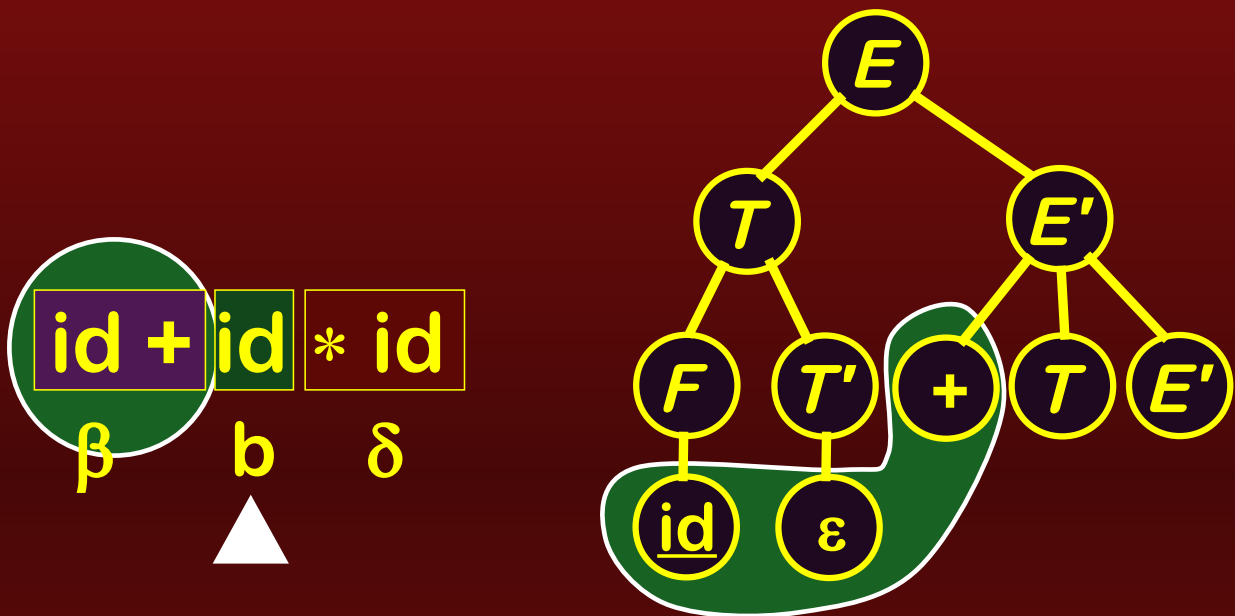
The **leaves** at any point form a string
 $\beta A \gamma$

44



β only contains terminals

45



Next token is **b**

46

LL(1) Table Construction

- Consider the state

$$S \rightarrow^* \beta A \gamma$$

with **b** the next token and we are trying to match **$\beta b \gamma$**

- There are two possibilities

47

LL(1) Table Construction

- Consider the state
$$S \rightarrow^* \beta A \gamma$$
with **b** the next token and we are trying to match **$\beta b \gamma$**
- There are **two** possibilities

48

LL(1) Table Construction

1. **b** belongs to an expansion of **A**

Any $A \rightarrow \alpha$ can be used if **b** can start a string derived from α

49

LL(1) Table Construction

1. **b** belongs to an expansion of **A**

Any **A** $\rightarrow$ **α** can be used if **b** can **start** a string derived from **α**

50

LL(1) Table Construction

In this case we say that
 $b \in \text{FIRST}(\alpha)$

51

LL(1) Table Construction

2. **b** does not belong to an expansion of **A**

Expansion of **A** is empty, i.e.,

$A \rightarrow \varepsilon$ and **b** belongs an expansion of γ , e.g., $b\omega$

52

LL(1) Table Construction

2. **b** does not belong to an expansion of **A**

Expansion of **A** is empty, i.e.,

$A \rightarrow \varepsilon$ and **b** belongs an expansion of γ , e.g., $b\omega$

53

LL(1) Table Construction

which means that **b** can appear **after A** in a derivation of the form

$$S \rightarrow^* \beta A b \omega$$

54

LL(1) Table Construction

We say that

$$b \in \text{FOLLOW}(A)$$

55

LL(1) Table Construction

Any $A \rightarrow \alpha$ can be used if α expands to ϵ

We say that $\epsilon \in \text{FIRST}(A)$ in this case

56

Computing FIRST Sets

Definition

$$\begin{aligned} \text{FIRST}(X) = & \\ & \{ b \mid X \rightarrow^* ba \} \\ & \cup \\ & \{ \epsilon \mid X \rightarrow^* \epsilon \} \end{aligned}$$

57

Compiler Construction

Sohail Aslam

Lecture 18

Computing FIRST Sets

For terminal symbols,

$$\text{FIRST}(b) = \{b\}$$

Computing FIRST Sets

For all productions

$$X \rightarrow A_1 \dots A_n$$

- Add $\text{FIRST}(A_1) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_1)$
- Add $\text{FIRST}(A_2) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_2)$

3

Computing FIRST Sets

For all productions

$$X \rightarrow A_1 \dots A_n$$

- Add $\text{FIRST}(A_1) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_1)$
- Add $\text{FIRST}(A_2) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_2)$

4

Computing FIRST Sets

For all productions

$$X \rightarrow A_1 \dots A_n$$

- Add $\text{FIRST}(A_1) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_1)$
- Add $\text{FIRST}(A_2) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_2)$

5

Computing FIRST Sets

-
- Add $\text{FIRST}(A_n) - \{\epsilon\}$ to $\text{FIRST}(X)$, stop if $\epsilon \notin \text{FIRST}(A_n)$
- Add ϵ to $\text{FIRST}(X)$

6

for each $a \in (T \cup \varepsilon)$

FIRST(a) $\leftarrow \{a\}$

for each $A \in NT$

FIRST(A) $\leftarrow \emptyset$

while (**FIRST** sets are still changing)

for each $A \rightarrow \beta_1 \beta_2 \dots \beta_k \in P$

FIRST(A) $\leftarrow \text{FIRST}(A) \cup (\text{FIRST}(\beta_1) - \{\varepsilon\})$

$i \leftarrow 1$

while ($\varepsilon \in \text{FIRST}(\beta_i)$ and $i \leq k-1$)

FIRST(A) $\leftarrow \text{FIRST}(A) \cup (\text{FIRST}(\beta_{i+1}) - \{\varepsilon\})$

$i \leftarrow i+1$

if ($i == k$ and $\varepsilon \in \text{FIRST}(\beta_k)$)

FIRST(A) $\leftarrow \text{FIRST}(A) \cup \{\varepsilon\}$

7

Computing FIRST Sets

the expression grammar again

1	E	$\rightarrow$	$T E'$
2	E'	$\rightarrow$	$+ T E'$
3		$ $	ε
4	T	$\rightarrow$	$F T'$
5	T'	$\rightarrow$	$* F T'$
6		$ $	ε
7	F	$\rightarrow$	(E)
8		$ $	<u>id</u>

8

Computing FIRST Sets

$$\begin{array}{ll} E & \rightarrow T E' \\ E' & \rightarrow + T E' \mid \varepsilon \\ T & \rightarrow F T' \\ T' & \rightarrow * F T' \mid \varepsilon \\ F & \rightarrow (E) \mid \underline{\text{id}} \end{array}$$

$\text{FIRST}(\underline{\text{id}}) = \{ \underline{\text{id}} \}$

$\text{FIRST}('(') = \{ (\}$

$\text{FIRST}('+') = \{ + \}$

9

Computing FIRST Sets

$$\begin{array}{ll} E & \rightarrow T E' \\ E' & \rightarrow + T E' \mid \varepsilon \\ T & \rightarrow F T' \\ T' & \rightarrow * F T' \mid \varepsilon \\ F & \rightarrow (E) \mid \underline{\text{id}} \end{array}$$

$\text{FIRST}(E) = \{ \text{FIRST}(T) - \{ \varepsilon \} \}$

$\text{FIRST}(T) = \{ \text{FIRST}(F) - \{ \varepsilon \} \}$

$\text{FIRST}(F) = \{ \text{FIRST}('(') - \{ \varepsilon \} \}$
 $= \{ (\}$

10

Computing FIRST Sets

$$\begin{array}{ll} E & \rightarrow T E' \\ E' & \rightarrow + T E' \mid \varepsilon \\ T & \rightarrow F T' \\ T' & \rightarrow * F T' \mid \varepsilon \\ F & \rightarrow (E) \mid \underline{\text{id}} \end{array}$$

$$\begin{aligned} \text{FIRST}(F) &= \\ &= \{ ' (' \} + \{ \text{FIRST}(\underline{\text{id}}) - \{ \varepsilon \} \} \\ &= \{ (, \underline{\text{id}} \} \end{aligned}$$

11

Computing FIRST Sets

$$\begin{array}{ll} E & \rightarrow T E' \\ E' & \rightarrow + T E' \mid \varepsilon \\ T & \rightarrow F T' \\ T' & \rightarrow * F T' \mid \varepsilon \\ F & \rightarrow (E) \mid \underline{\text{id}} \end{array}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

12

Computing FIRST Sets

Thus,

- $\text{FIRST}(E) =$
 $\text{FIRST}(T) =$
 $\text{FIRST}(F) = \{ (, \underline{\text{id}} \}$
- $\text{FIRST}(E') = \{ +, \varepsilon \}$
- $\text{FIRST}(T') = \{ *, \varepsilon \}$

13

FOLLOW Sets

Definition:

$$\text{FOLLOW}(X) = \{ b \mid S \xrightarrow{*} \beta X b \omega \}$$

14

Computing FOLLOW Sets

1. Add $\$$ to **FOLLOW(S)**
where **S** is the start non-terminal

15

Computing FOLLOW Sets

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in **FIRST(β)** – $\{\epsilon\}$ is in **FOLLOW(B)**

16

Computing FOLLOW Sets

3. If there is a production
 $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$,
where in $\varepsilon \in \text{FIRST}(\beta)$
(i.e., $\beta \rightarrow^* \varepsilon$), then
everything in $\text{FOLLOW}(A)$
is in $\text{FOLLOW}(B)$

17

```
for each  $A \in NT$ 
   $\text{FOLLOW}(A) \leftarrow \emptyset$ 
 $\text{FOLLOW}(S) \leftarrow \{\$$ 
while (  $\text{FOLLOW}$  sets are still changing)
  for each  $A \rightarrow \beta_1 \beta_2 \dots \beta_k \in P$ 
     $\text{FOLLOW}(\beta_k) \leftarrow \text{FOLLOW}(\beta_k) \cup \text{FOLLOW}(A)$ 
     $T \leftarrow \text{FOLLOW}(A)$ 
    for  $i \leftarrow k$  downto 2
      if (  $\varepsilon \in \text{FIRST}(\beta_i)$  )
         $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup$ 
         $(\text{FIRST}(\beta_i) - \{\varepsilon\}) \cup T$ 
      else
         $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup \text{FIRST}(\beta_i)$ 
     $T \leftarrow \emptyset$ 
```

18

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & + T E' \mid \varepsilon \\
 T & \rightarrow & F T' \\
 T' & \rightarrow & * F T' \mid \varepsilon \\
 F & \rightarrow & (E) \mid \underline{\text{id}}
 \end{array}$$

Put **\$** in **FOLLOW(*E*)**

19

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & + T E' \mid \varepsilon \\
 T & \rightarrow & F T' \\
 T' & \rightarrow & * F T' \mid \varepsilon \\
 F & \rightarrow & (E) \mid \underline{\text{id}}
 \end{array}$$

By rule (2) applied to production $F \rightarrow (E)$, **') '** is also in **FOLLOW(*E*)**

20

$$\begin{array}{ll}
 E & \rightarrow T E' \\
 E' & \rightarrow + T E' \mid \varepsilon \\
 T & \rightarrow F T' \\
 T' & \rightarrow * F T' \mid \varepsilon \\
 F & \rightarrow (E) \mid \underline{\text{id}}
 \end{array}$$

Thus, **FOLLOW**(E) = { $)$, $\$$ }

21

$$\begin{array}{ll}
 E & \rightarrow T E' \\
 E' & \rightarrow + T E' \mid \varepsilon \\
 T & \rightarrow F T' \\
 T' & \rightarrow * F T' \mid \varepsilon \\
 F & \rightarrow (E) \mid \underline{\text{id}}
 \end{array}$$

By rule (3) applied to production $E \rightarrow T E'$, $\$$ and $)$ are in **FOLLOW**(E')

22

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & + T E' \mid \varepsilon \\
 T & \rightarrow & F T' \\
 T' & \rightarrow & * F T' \mid \varepsilon \\
 F & \rightarrow & (E) \mid \underline{\text{id}}
 \end{array}$$

thus,

$$\text{FOLLOW}(E) =$$

$$\text{FOLLOW}(E') = \{), \$ \}$$

23

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & + T E' \mid \varepsilon \\
 T & \rightarrow & F T' \\
 T' & \rightarrow & * F T' \mid \varepsilon \\
 F & \rightarrow & (E) \mid \underline{\text{id}}
 \end{array}$$

Similarly,

$$\text{FOLLOW}(T) =$$

$$\text{FOLLOW}(T') = \{ +,), \$ \}$$

24

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & + T E' \mid \varepsilon \\
 T & \rightarrow & F T' \\
 T' & \rightarrow & * F T' \mid \varepsilon \\
 F & \rightarrow & (E) \mid \underline{\text{id}}
 \end{array}$$

and

$$\text{FOLLOW}(F) = \{ +, *,), \$ \}$$

25

LL(1) Table Construction

Here now is the algorithm
to **construct** a predictive
parsing **table**.

26

Compiler Construction

Sohail Aslam

Lecture 19

LL(1) Table Construction

For each production $A \rightarrow \alpha$

1. for each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$.

LL(1) Table Construction

For each production $A \rightarrow \alpha$

1. for each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$.

3

LL(1) Table Construction

For each production $A \rightarrow \alpha$

1. for each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$.

4

LL(1) Table Construction

(2 cont.)

If ϵ is in **FIRST**(α), and $\$$ is in **FOLLOW**(A), add $A \rightarrow \alpha$ to **M**[$A, \$$].

5

LL(1) Table Construction

Make each undefined entry of **M** be error.

6

LL(1) Table Construction

Let us apply the algorithm to the expression grammar.

7

LL(1) Table Construction

Since

$\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, \underline{\text{id}} \}$,
production $E \rightarrow TE'$ cause $M[E, (]$ and
 $M[E, \underline{\text{id}}]$ to get $E \rightarrow TE'$

8

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \varepsilon$	$T \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
<i>F</i>	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		



9

LL(1) Table Construction

Production $E' \rightarrow +TE'$ causes

$M[E', +]$ to get $E' \rightarrow +TE'$

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \varepsilon$	$T \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
<i>F</i>	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		



11

LL(1) Table Construction

Production $E' \rightarrow \varepsilon$ causes $M[E',)]$ and $M[E', \$]$ to get $E' \rightarrow \varepsilon$ since $\text{FOLLOW}(E') = \{), \$ \}$

12

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \varepsilon$	$T \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
<i>F</i>	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		



13

LL(1) Table Construction

And so on.

The final parsing table produced is:

LL(1) Parsing Table

	<u>id</u>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \epsilon$	$T \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<i>F</i>	$F \rightarrow \underline{\text{id}}$			$F \rightarrow (E)$		

15

Left Factoring

Consider the grammar

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

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

16

Left Factoring

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

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$



Impossible to predict because for **T**,
two productions start with **int**

17

Left Factoring

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

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$



For **E**, it is not clear how to predict

18

Left Factoring

A grammar must be **left factored** before use for predictive parsing

19

Left Factoring

If $\alpha \neq \varepsilon$, replace all productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha Z \mid \gamma$$

$$\square Z \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where Z is new nonterminal

20

Left Factoring

If $\alpha \neq \varepsilon$, replace all productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha Z \mid \gamma$$

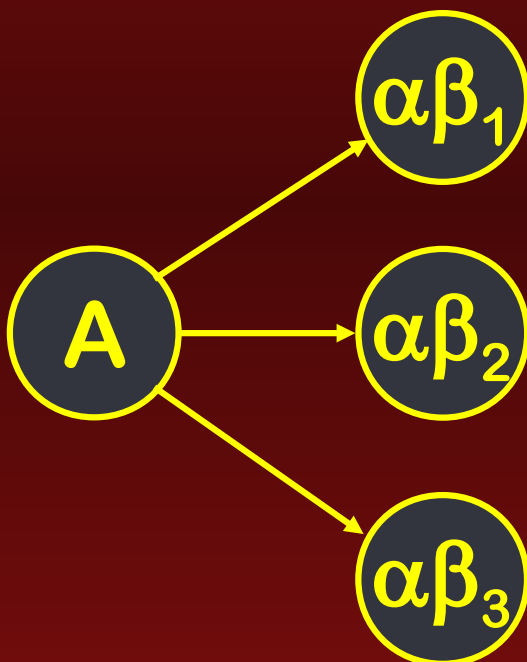
$$\square Z \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where Z is new nonterminal

21

Left Factoring

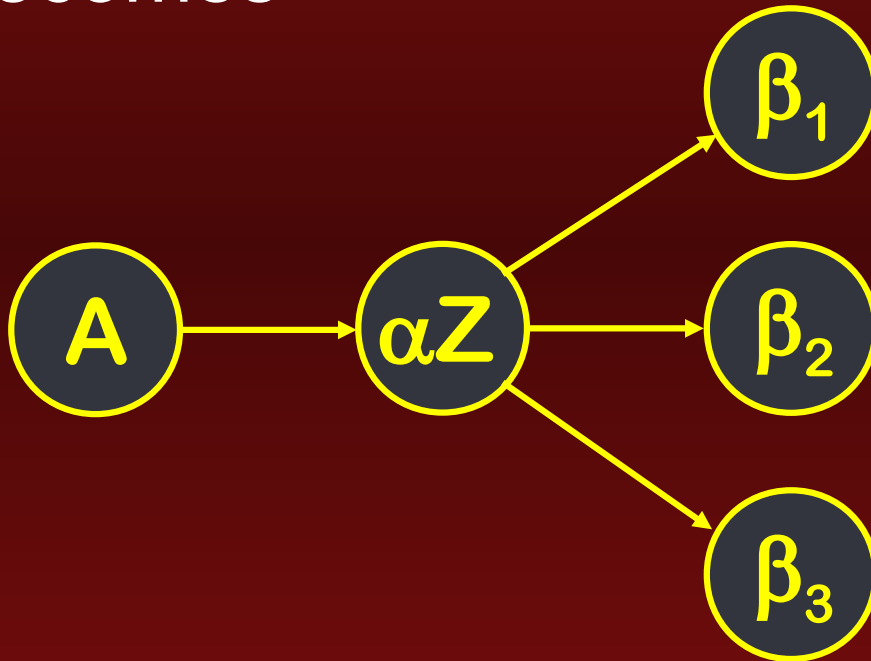
A graphical explanation



22

Left Factoring

becomes



23

Left Factoring

Consider following fragment of expression grammar

Factor $\rightarrow$ id
 | id [ExprList]
 | id (ExprList)

24

Left Factoring

Factor $\rightarrow$ id
| id [ExprList]
| id (ExprList)

FIRST(rhs₁) = {id}

FIRST(rhs₂) = {id}

FIRST(rhs₃) = {id}

25

Left Factoring

after left factoring

◆ Factor $\rightarrow$ id Args

Args $\rightarrow$ [ExprList]
| (ExprList)
| ϵ

26

Left Factoring

Factor $\rightarrow$ id Args
Args $\rightarrow$ [ExprList]
 | (ExprList)
 | ε

$\text{FIRST}(\text{rhs}_1) = \{\text{id}\}$

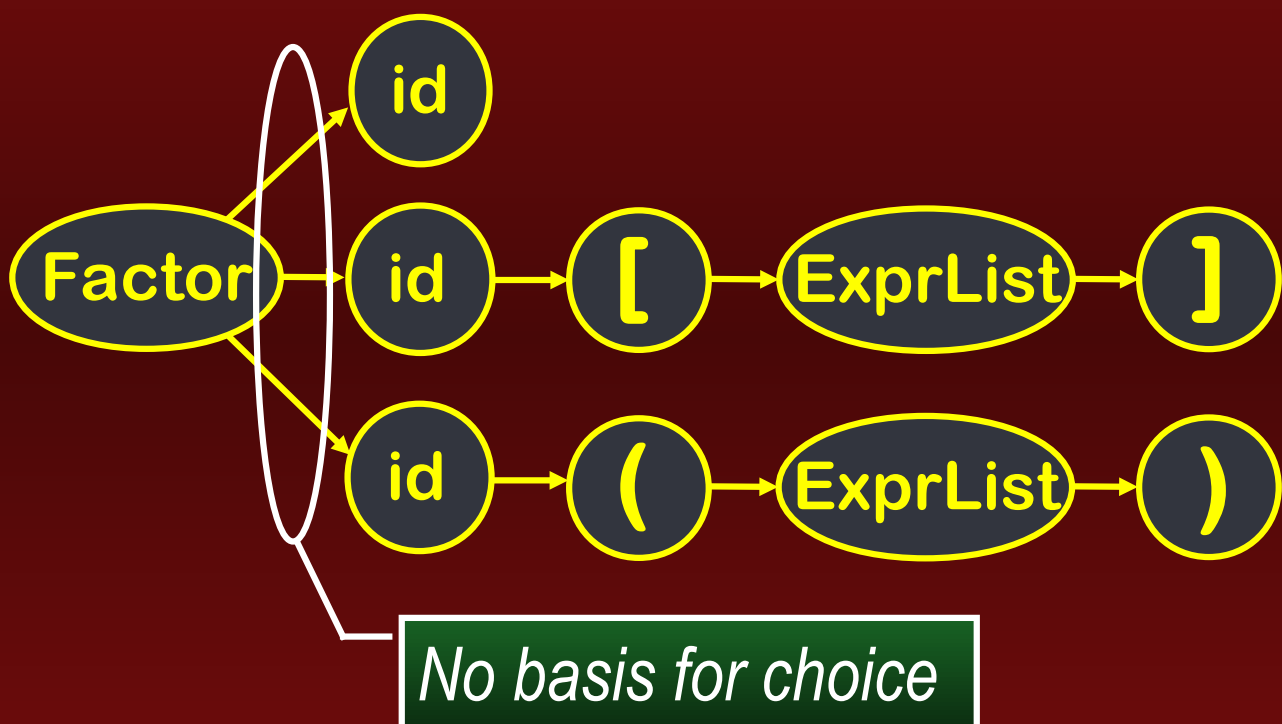
$\text{FIRST}(\text{rhs}_2) = \{ [\}$

$\text{FIRST}(\text{rhs}_3) = \{ (\}$

$\text{FIRST}(\text{rhs}_4) = \text{FOLLOW}(\text{Factor})$

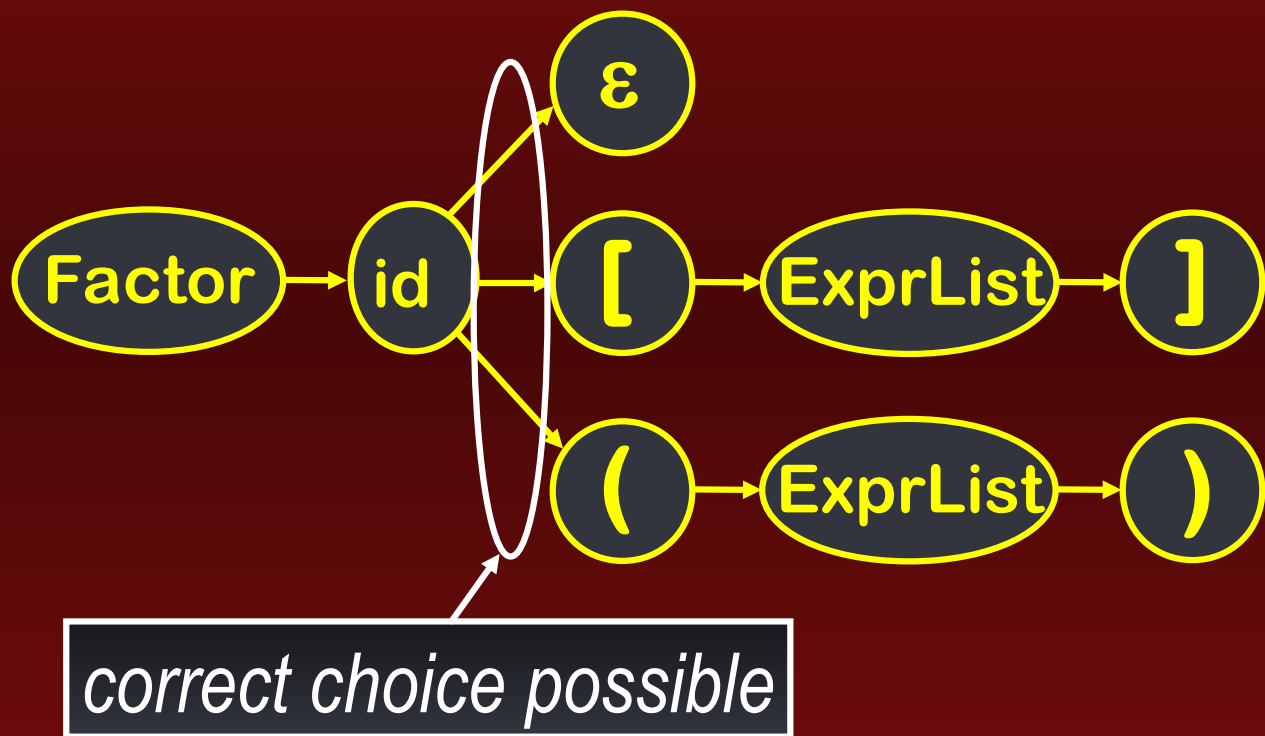
27

Left Factoring



28

Left Factoring



29

Left Factoring

Question

By eliminating left recursion and left factoring, can we transform an arbitrary CFG to a form where it meets the LL(1) condition (and can be parsed predictively with a single token lookahead?)

30

Left Factoring

Answer

Given a CFG that does not meet the LL(1) condition, it is *undecidable* whether or not an equivalent LL(1) grammar exists

31

Compiler Construction

Sohail Aslam

Lecture 20

Bottom-up Parsing

- Bottom-up parsing is **more general** than top-down parsing
- Bottom-up parsers handle a large class of grammars.
- Preferred method in practice

2

Bottom-up Parsing

- Bottom-up parsing is **more general** than top-down parsing
- Bottom-up parsers handle a **large class** of grammars.
- Preferred method in practice

3

Bottom-up Parsing

- Bottom-up parsing is **more general** than top-down parsing
- Bottom-up parsers handle a **large class** of grammars.
- **Preferred** method in practice

4

Bottom-up Parsing

Also called **LR** parsing

- L means that tokens are read left to right
- R means that the parser constructs a rightmost derivation.

5

Bottom-up Parsing

Also called LR parsing

- L means that tokens are read left to right
- R means that the parser constructs a rightmost derivation.

6

Bottom-up Parsing

Also called LR parsing

- L means that tokens are read left to right
- R means that the parser constructs a rightmost derivation.

7

Bottom-up Parsing

- LR parsers **do not** need left-factored grammars
- LR parsers can handle left-recursive grammars

8

Bottom-up Parsing

- LR parsers **do not** need left-factored grammars
- LR parsers **can** handle left-recursive grammars

9

Bottom-up Parsing

- LR parsing *reduces* a string to the start symbol by *inverting* productions.

10

Bottom-up Parsing

- A derivation consists of a series of rewrite steps

$$\begin{aligned} \mathbf{S} &\Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_{n-1} \\ &\Rightarrow \gamma_n \Rightarrow \mathbf{sentence} \end{aligned}$$

11

Bottom-up Parsing

$S \Rightarrow \gamma_0 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \text{sentence}$

Each γ_i is a sentential form

- if γ contains only terminals, γ is a sentence in $L(G)$
- If γ contains ≥ 1 nonterminals, γ is a sentential form

12

Bottom-up Parsing

$S \Rightarrow \gamma_0 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \text{sentence}$

Each γ_i is a sentential form

- if γ contains only terminals, γ is a sentence in $L(G)$
- If γ contains ≥ 1 nonterminals, γ is a sentential form

13

Bottom-up Parsing

$S \Rightarrow \gamma_0 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \text{sentence}$

Each γ_i is a sentential form

- if γ contains only terminals, γ is a sentence in $L(G)$
- If γ contains ≥ 1 nonterminals, γ is a sentential form

14

Bottom-up Parsing

- A bottom-up parser builds a derivation by working from input sentence back towards the start symbol S

$S \Rightarrow \gamma_0 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \text{sentence}$



15

Bottom-up Parsing

- Consider the grammar

S $\rightarrow$ **aABe**

A $\rightarrow$ **Abc | b**

B $\rightarrow$ **d**

16

Bottom-up Parsing

- The sentence **abbcde** can be reduced to **S**:

abbcde

aAbcde

aAde

aABe

S

17

Bottom-up Parsing

- The sentence **abbcede** can be reduced to **S**:

abbcede

aAbcde

aAde

aABe

S

18

Bottom-up Parsing

- The sentence **abbcede** can be reduced to **S**:

abbcede

aAbcde

aAde

aABe

S

19

Bottom-up Parsing

- The sentence **abbcede** can be reduced to **S**:

abbcede

aAbcde

aAde

aABe

S

20

Bottom-up Parsing

- The sentence **abbcede** can be reduced to **S**:

abbcede

aAbcde

aAde

aABe

S

21

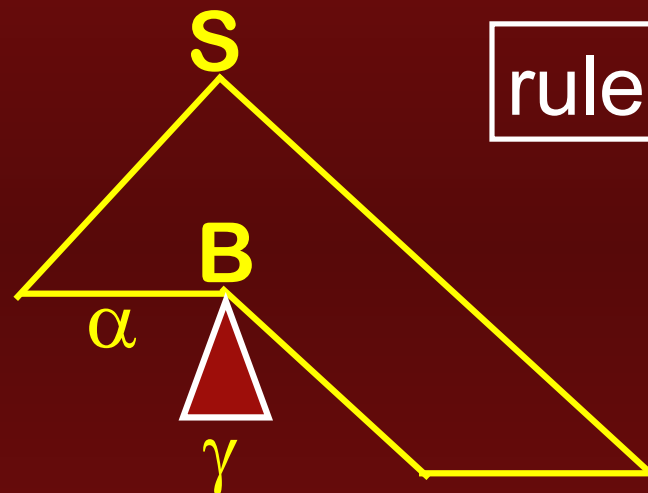
Bottom-up Parsing

- These reductions, in fact, trace out the following **right-most** derivation in **reverse**:

$S \Rightarrow aABe$
 $\Rightarrow aAde$
 $\Rightarrow aAbcde$
 $\Rightarrow abbcde$

22

$S \xRightarrow{rm}^* \alpha \mathbf{B} \mathbf{y} \xRightarrow{rm} \alpha \mathbf{\gamma} \mathbf{y} \xRightarrow{rm}^* \mathbf{x} \mathbf{y}$



rule: $B \rightarrow \gamma$



Terminals only

23

Bottom-up Parsing

Consider the grammar

1. $E \rightarrow E + (E)$
2. $E \rightarrow \underline{\text{int}}$

24

Bottom-up Parsing

Consider bottom-up parse of the string

$\text{int} + (\text{int}) + (\text{int})$

25

int + (int) + (int)

int + (int) + (int)

26

int + (int) + (int)

E + (int) + (int)

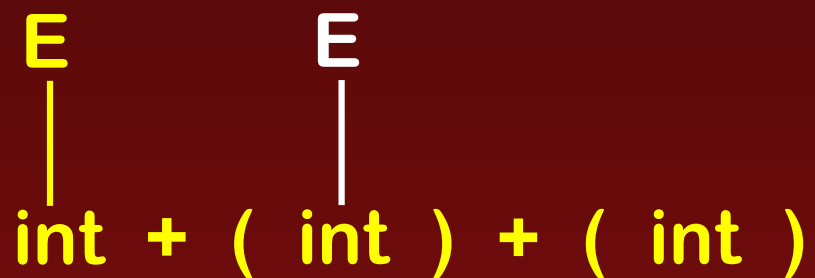
E
|
int + (int) + (int)

27

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



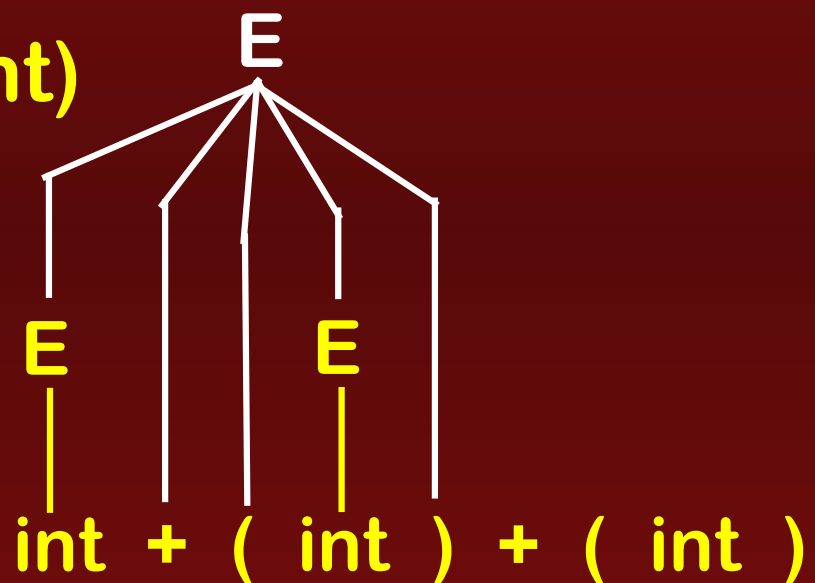
28

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



29

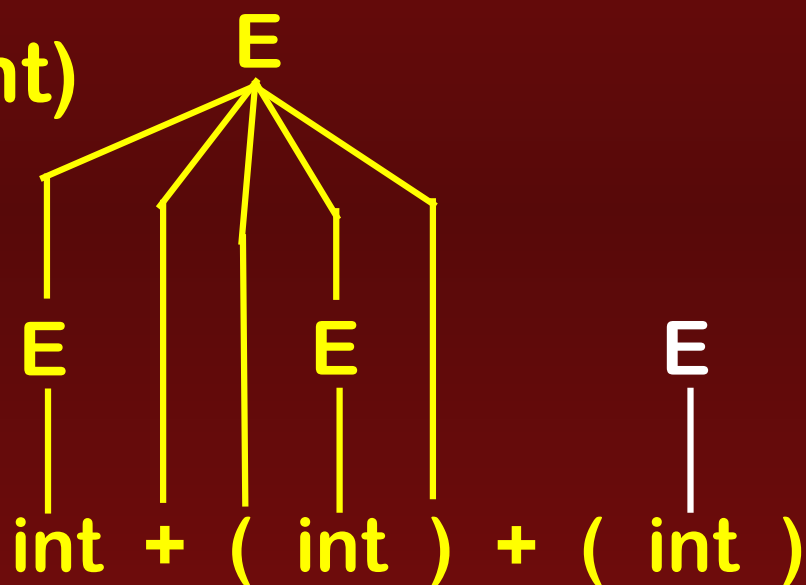
$\text{int} + (\text{int}) + (\text{int})$

$E + (\text{int}) + (\text{int})$

$E + (E) + (\text{int})$

$E + (\text{int})$

$E + (E)$



30

A rightmost derivation in reverse

$\uparrow$ $\text{int} + (\text{int}) + (\text{int})$

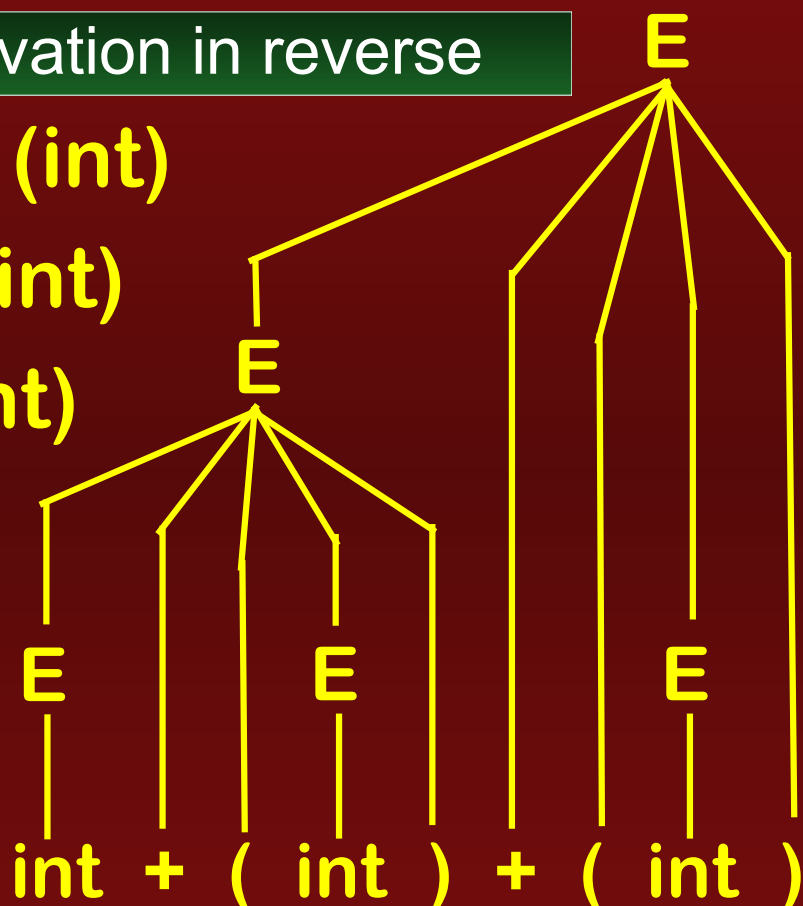
$E + (\text{int}) + (\text{int})$

$E + (E) + (\text{int})$

$E + (\text{int})$

$E + (E)$

E



31

Bottom-up Parsing

An LR parser traces a rightmost derivation in reverse

32

Consequence

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume that next reduction is $A \rightarrow \beta$
- Then γ is a string of terminals!

33

Consequence

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume that next reduction is $A \rightarrow \beta$
- Then γ is a string of terminals!

34

Consequence

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume that next reduction is $A \rightarrow \beta$
- Then γ is a string of terminals!

35

Consequence

Why?

- Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a rightmost derivation

36

Notation

Idea:

- Split the string into two substrings

37

Notation

- **Right** substring (a string of terminals) is as yet **unexamined** by parser
- Left substring has terminals and non-terminals

38

Notation

- **Right** substring (a string of terminals) is as yet **unexamined** by parser
- **Left** substring has terminals and non-terminals

39

Notation

- The **dividing point** is marked by a ►
 - *The ► is not part of the string*
- Initially, all **input** is unexamined: ► $x_1 x_1 \dots x_n$

40

Shift-Reduce Parsing

Bottom-up parsing uses only **two** kinds of **actions**:

- 1. Shift**
- 2. Reduce**

41

Shift

Move ► one place to the right
shifts a terminal to the left
string

$$E + (\blacktriangleright \text{int}) \Rightarrow E + (\text{int} \blacktriangleright)$$

↑

42

Reduce

Apply an inverse production
at the right end of the left
string. If $E \rightarrow E + (E)$ is a
production, then

$$E + (E + (E) \blacktriangleright) \Rightarrow E + (E \blacktriangleright)$$

43

Shift-Reduce Example

► **int + (int) + (int) \$** shift

int ► **+** (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► **+** (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► **+** (int) \$ reduce $E \rightarrow E+(E)$

44

Shift-Reduce Example

► **int + (int) + (int) \$** shift

int ► **+** (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► **+** (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► **+** (int) \$ reduce $E \rightarrow E+(E)$

45

Shift-Reduce Example

► **int + (int) + (int) \$** shift

int ► **+ (int) + (int) \$** reduce **E** → **int**

E ► **+ (int) + (int) \$** shift 3 times

E + (int ► **) + (int) \$** reduce **E** → **int**

E + (E ► **) + (int) \$** shift

E + (E) ► **+ (int) \$** reduce **E** → **E+(E)**

46

Shift-Reduce Example

► **int + (int) + (int) \$** shift

int ► **+ (int) + (int) \$** reduce **E** → **int**

E ► **+ (int) + (int) \$** shift 3 times

E + (int ► **) + (int) \$** reduce **E** → **int**

E + (E ► **) + (int) \$** shift

E + (E) ► **+ (int) \$** reduce **E** → **E+(E)**

47

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

48

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

49

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

50

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

51

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

52

Shift-Reduce Example

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$

E + (E ►) + (int) \$ shift

E + (E) ► + (int) \$ reduce $E \rightarrow E+(E)$

53

Shift-Reduce Example

→ E ► + (int) \$	shift 3 times
E + (int ►) \$	reduce E → int
E + (E ►) \$	shift
E + (E) ► \$	red E → E+(E)
E ► \$	<u>accept</u>

54

Compiler Construction

Sohail Aslam

Lecture 21

Shift-Reduce: The Stack

- Left string can implemented as a stack
- Top of the stack is the ►
- Shift pushes a terminal on the stack

2

Shift-Reduce: The Stack

- Left string can implemented as a stack
- Top of the stack is the ►
- Shift pushes a terminal on the stack

3

Shift-Reduce: The Stack

- Left string can implemented as a stack
- Top of the stack is the ►
- Shift pushes a terminal on the stack

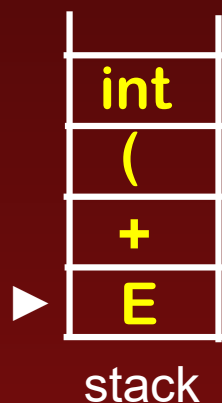
4

► int + (int) + (int) \$ shift

int ► + (int) + (int) \$ reduce $E \rightarrow \text{int}$

E ► + (int) + (int) \$ shift 3 times

E + (int ►) + (int) \$ reduce $E \rightarrow \text{int}$



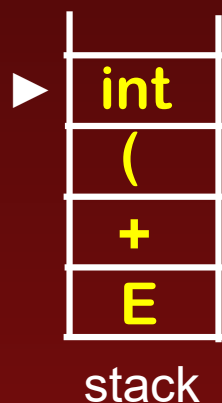
5

Shift-Reduce: The Stack

- Reduce **pops** zero or more symbols from the stack (production *rhs*) and **pushes** a **non-terminal** on the stack (production *lhs*)

6

► **int + (int) + (int) \$** shift
int ► **+ (int) + (int) \$** reduce **E** → **int**
E ► **+ (int) + (int) \$** shift 3 times
E + (int ► **) + (int) \$** reduce **E** → **int**



7

Discovering Handles

- A bottom-up parser builds the **parse tree** starting with its **leaves** and working **toward its root**

8

Discovering Handles

- The upper edge of this partially constructed parse tree is called its **upper frontier**.

9

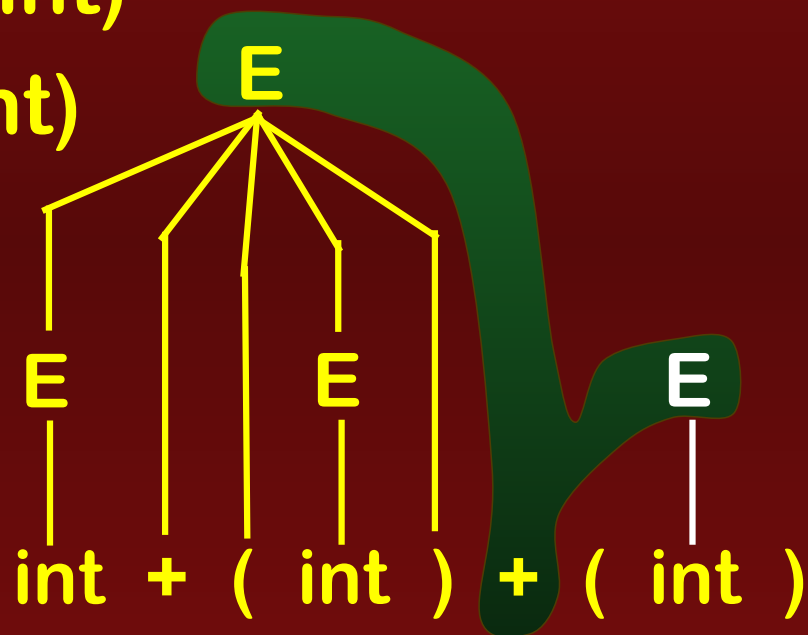
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)



10

Discovering Handles

- At each step, the parser looks for a section of the upper frontier that matches right-hand side of some production.

11

Discovering Handles

- When it finds a match, the parser builds a new tree node with the production's left-hand non-terminal thus extending the frontier upwards towards the root.

12

Discovering Handles

- The critical step is developing an efficient mechanism that finds matches along the tree's current frontier.

13

Discovering Handles

Formally, the parser must find some substring β , of the upper frontier where

1. β is the right-hand side of some production $A \rightarrow \beta$, and
2. $A \rightarrow \beta$ is one step in right-most derivation of input stream

14

Discovering Handles

Formally, the parser must find some substring β , of the upper frontier where

1. β is the right-hand side of some production $A \rightarrow \beta$, and
2. $A \rightarrow \beta$ is one step in right-most derivation of input stream

15

Discovering Handles

Formally, the parser must find some substring β , of the upper frontier where

1. β is the right-hand side of some production $A \rightarrow \beta$, and
2. $A \rightarrow \beta$ is one step in right-most derivation of input stream

16

Discovering Handles

- We can represent each potential match as a pair $\langle A \rightarrow \beta, k \rangle$, where k is the position on the tree's current frontier of the right-end of β .

17

Discovering Handles

- The pair $\langle A \rightarrow \beta, k \rangle$, is called the **handle** of the bottom-up parse.

18

Handle Pruning

A bottom-up parser **operates** by repeatedly **locating handles** on the frontier of the partial parse tree and performing **reductions** that they specify.

19

Handles

- The bottom-up parser uses a **stack** to hold the frontier.
- The stack simplifies the parsing algorithm in two ways.

20

Handles

- The bottom-up parser uses a stack to hold the frontier.
- The stack **simplifies** the parsing algorithm in **two** ways.

21

Handles

First, the stack trivializes the problem of managing space for the frontier

- To extend the frontier, the parser simply pushes the current input token onto the top of the stack

22

Handles

First, the stack trivializes the problem of managing space for the frontier

- To extend the frontier, the parser simply pushes the current input token onto the top of the stack

23

Handles

Second, the stack ensures that **all handles** occur with their **right end** at the **top** of the stack

- eliminates the need to represent handle's position

24

Handles

Second, the stack ensures that **all handles** occur with their **right end** at the **top** of the stack

- eliminates the need to represent handle's position

25

→ $E \triangleright + (int) \$$

$E \triangleright + (int) \$$
1 2 3 4

$E \triangleright + (E) \$$

$E \triangleright + (E) \$$

$E \triangleright \$$

shift 3 times

reduce $E \rightarrow int$

shift

red $E \rightarrow E+(E)$

accept

handle: $\langle E \rightarrow E+(E), 5 \rangle$

26

Shift-Reduce Parsing Algorithm

push $\$$ onto stack

$sym \leftarrow \text{nextToken}()$

repeat until ($sym == \$$ and the stack contains exactly **Goal** on top of $\$$)

if a handle for $A \rightarrow \beta$ on top of stack

pop $|\beta|$ symbols off the stack

push A onto the stack

27

Shift-Reduce Parsing Algorithm

else if (sym $\neq$ \$)

push sym onto stack

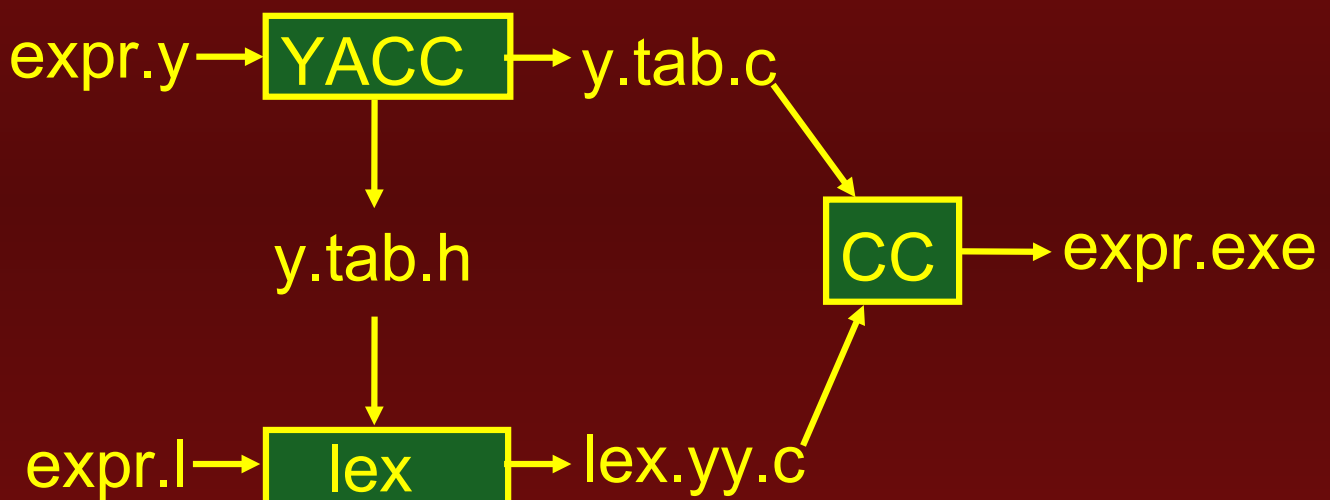
sym $\leftarrow$ nextToken()

else /* no handle, no input */

report error and halt

28

Building a parser with YACC and Lex



29

Compiler Construction

Sohail Aslam

Lecture 22

Expression Grammar

1	Goal	→	Expr
2	Expr	→	Expr + Term
3			Expr – Term
4			Term
5	Term	→	Term × Factor
6			Term / Factor
7			Factor
8	Factor	→	<u>num</u>
9			<u>id</u>
10			(Expr)

The states of the bottom-up parser on input

x – 2 × y

(tokenized as **id – num * id**) are

3

	word	Stack	Handle	Action
1	id	►	- none -	<i>shift</i>
2	–	id ►	⟨ Factor → id ,1⟩	<i>reduce</i>
3	–	Factor ►	⟨ Term → Factor ,1⟩	<i>reduce</i>
4	–	Term ►	⟨ Expr → Term ,1⟩	<i>reduce</i>
5	–	Expr ►	- none -	<i>shift</i>
6	num	Expr – ►	- none -	<i>shift</i>
7	×	Expr – num ►	⟨ Factor → num ,3⟩	<i>reduce</i>
8	×	Expr – Factor ►	⟨ Term → Factor ,3⟩	<i>shift</i>

4

	word	Stack	Handle	Action
9	×	Expr – Term ►	- none -	<i>shift</i>
10	id	Expr – Term × ►	- none -	<i>shift</i>
11	\$	Expr – Term × id ►	⟨Factor → id,5⟩	<i>reduce</i>
12	\$	Expr – Term × Factor ►	⟨Term → Term × Factor,5⟩	<i>reduce</i>
13	\$	Expr – Term ►	⟨Expr → Expr – Term,3⟩	<i>reduce</i>
14	\$	Expr ►	⟨Goal → Expr,1⟩	<i>reduce</i>
15	\$	Goal	- none -	<u><i>accept</i></u>

5

Handles

- The **handle-finding** mechanism is the **key** to efficient **bottom-up parsing**
- As it process an input string, the parser must find and track all potential handles

6

Handles

- The handle-finding mechanism is the key to efficient bottom-up parsing
- As it process an input string, the parser must find and track all potential handles

7

Handles

- For example, every legal input eventually reduces the entire frontier to grammar's goal symbol
- Thus, $\langle \text{Goal} \rightarrow \text{Expr}, 1 \rangle$ is a potential handle at the start of every parse

8

Handles

- For example, every legal input eventually reduces the entire frontier to grammar's goal symbol
- Thus, $\langle \text{Goal} \rightarrow \text{Expr}, 1 \rangle$ is a potential handle at the start of every parse

9

Handles

- As the parser builds a derivation, it discovers other handles
- At each step, the set of potential handles represent different suffixes that lead to a reduction.

10

Handles

- As the parser builds a derivation, it discovers other handles
- At **each step**, the set of potential handles represent **different suffixes** that lead to a reduction.

11

Handles

- Each **potential handle** represent a **string** of **grammar symbols** that, if seen, would **complete** the **right-hand** side of some **production**.

12

Handles

- For the bottom-up parse of the **expression grammar string**, we can represent the **potential handles** that the shift-reduce parser should **track**

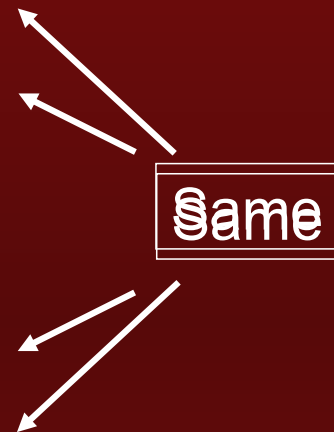
13

Handles

Using the placeholder **•** to represent **top of the stack**, there are **nine** handles:

14

	Handles
1	$\langle \text{Factor} \rightarrow \text{id} \bullet \rangle$
2	$\langle \text{Term} \rightarrow \text{Factor} \bullet \rangle$
3	$\langle \text{Expr} \rightarrow \text{Term} \bullet \rangle$
4	$\langle \text{Factor} \rightarrow \text{num} \bullet \rangle$
5	$\langle \text{Term} \rightarrow \text{Factor} \bullet \rangle$
6	$\langle \text{Factor} \rightarrow \text{id} \bullet \rangle$
7	$\langle \text{Term} \rightarrow \text{Term} \times \text{Factor} \bullet \rangle$
8	$\langle \text{Expr} \rightarrow \text{Expr} - \text{Term} \bullet \rangle$
9	$\langle \text{Goal} \rightarrow \text{Expr} \bullet \rangle$



15

Handles

- This notation shows that the second and fifth handles are identical, as are first and sixth
- It also create a way to represent the potential of discovering a handle in future.

16

Handles

- This notation shows that the second and fifth handles are identical, as are first and sixth
- It **also** create a way to represent the **potential** of discovering a handle in future.

17

Handles

Consider the parser's state in step 6:

→	6	num	Expr – ►	- none -	shift
	7	×	Expr – num ►	⟨Factor → num,3⟩	reduce
	8	×	Expr – Factor ►	⟨Term → Factor,3⟩	shift
	9	×	Expr – Term ►	- none -	shift
	10	id	Expr – Term × ►	- none -	shift
	11	\$	Expr – Term × id ►	⟨Factor → id,5⟩	reduce
	12	\$	Expr – Term × Factor ►	⟨Term → Term × Factor,5⟩	reduce
	13	\$	Expr – Term ►	⟨Expr → Expr – Term,3⟩	reduce

18

Handles

It needs to recognize, eventually, a **Term** before it can reduce this part of the frontier
 The parser has recognized **Expr –**

→	6	num	Expr – ►	- none -	shift
	7	×	Expr – num ►	⟨Factor → num,3⟩	reduce
	8	×	Expr – Factor ►	⟨Term → Factor,3⟩	shift
	9	×	Expr – Term ►	- none -	shift
	10	id	Expr – Term × ►	- none -	shift
	11	\$	Expr – Term × id ►	⟨Factor → id,5⟩	reduce
	12	\$	Expr – Term × Factor ►	⟨Term → Term × Factor,5⟩	reduce
	13	\$	Expr – Term ►	⟨Expr → Expr – Term,3⟩	reduce

19

Handles

- Using the stack-relative notation, we can represent the **parser's state** as **Expr → Expr – • Term**
- The parser has already recognized an Expr and a –

20

Handles

- Using the stack-relative notation, we can represent the parser's state as $\text{Expr} \rightarrow \text{Expr} - \bullet \text{Term}$
- The parser has already recognized an **Expr** and a **-**

21

Handles

- If the parser reaches a state where it shifts a **Term** on top of **Expr** and **-**, it will complete the handle $\text{Expr} \rightarrow \text{Expr} - \text{Term} \bullet$

22

Handles

- How **many** potential handles must the parser **recognize**?
- The right-hand side of each production can have a placeholder at its start, at its end and between any two consecutive symbols

23

Handles

- How many potential handles must the parser recognize?
- The **right-hand** side of each **production** can have a placeholder at its **start**, at its **end** and **between** any two consecutive **symbols**

24

Handles

$\text{Expr} \rightarrow \downarrow \bullet \text{Expr} - \text{Term}$

$\text{Expr} \rightarrow \text{Expr} \bullet - \text{Term}$

$\text{Expr} \rightarrow \text{Expr} - \bullet \text{Term}$

$\text{Expr} \rightarrow \text{Expr} - \text{Term} \bullet$

if the right-hand side has k symbols,
it has $k + 1$ placeholder positions

25

Compiler Construction

Sohail Aslam

Lecture 23

Handles

- The number of potential handles for the grammar is simply the sum of the lengths of the right-hand side of all the productions
- The number of complete handles is simply the number of productions

2

Handles

- The number of potential handles for the grammar is simply the sum of the lengths of the right-hand side of all the productions
- The number of complete handles is simply the number of productions

3

Handles

- These two facts lead to the critical insight behind LR parsers:

A given grammar generates a finite set of handles (and potential handles) that the parser must recognize

4

Handles

- These two facts lead to the critical insight behind LR parsers:

A given grammar generates a finite set of handles (and potential handles) that the parser must recognize

5

Lookahead

- However, it is not a simple matter of putting the placeholder in right-hand side to generate handles
- The parser needs to recognize the correct handle by different right contexts

6

Lookahead

- However, it is not a simple matter of putting the placeholder in right-hand side to generate handles
- The parser needs to **recognize** the correct handle by **different right contexts**

7

	word	Stack	Handle	Action
→ 9	×	Expr – Term ►	- none -	shift
10	id	Expr – Term × ►	- none -	shift
11	\$	Expr – Term × id ►	⟨Factor → id,5⟩	reduce
12	\$	Expr – Term × Factor ►	⟨Term → Term × Factor,5⟩	reduce
13	\$	Expr – Term ►	⟨Expr → Expr – Term,3⟩	reduce
14	\$	Expr ►	⟨Goal → Expr,1⟩	reduce
15	\$	Goal	- none -	<u>accept</u>

However, this parser decision stops finding the frontier **Expr** and the handle **Expr – Term** rather than reducing frontier to **Expr**

8

Lookahead

- Clearly, this the **correct move** for the **parser**
- No potential handle contains **Expr** followed by **x**

9

Lookahead

- Clearly, this the correct move for the parser
- No potential handle contains **Expr** followed by **x**

10

Lookahead

- At step 9, the set of potential handles is

$\langle \text{Expr} \rightarrow \text{Expr} - \text{Term} \bullet \rangle$

$\langle \text{Term} \rightarrow \text{Term} \bullet \times \text{Factor} \rangle$

$\langle \text{Term} \rightarrow \text{Term} \bullet / \text{Factor} \rangle$

11

Lookahead

- The next input symbol clearly matches the second choice

⟨Expr → Expr – Term•⟩

⟨Term → Term• × Factor⟩

⟨Term → Term• / Factor⟩

12

Lookahead

- The **parser** needs a basis for **deciding** between first (reduce) and second (shift) choices:

⟨Expr → Expr – Term•⟩

⟨Term → Term• × Factor⟩

13

Lookahead

- This requires **more context** than the parser has in the **frontier** (stack)

14

Lookahead

- To **choose** between reducing and shifting, the parser must **recognize** which **symbols** can occur to the **right** of **Expr** and **Term** in valid phrases

15

LR(1) Parsers

- The LR(1) parsers can recognize precisely those languages in which one-symbol lookahead suffices to determine whether to shift or reduce.

16

LR(1) Parsers

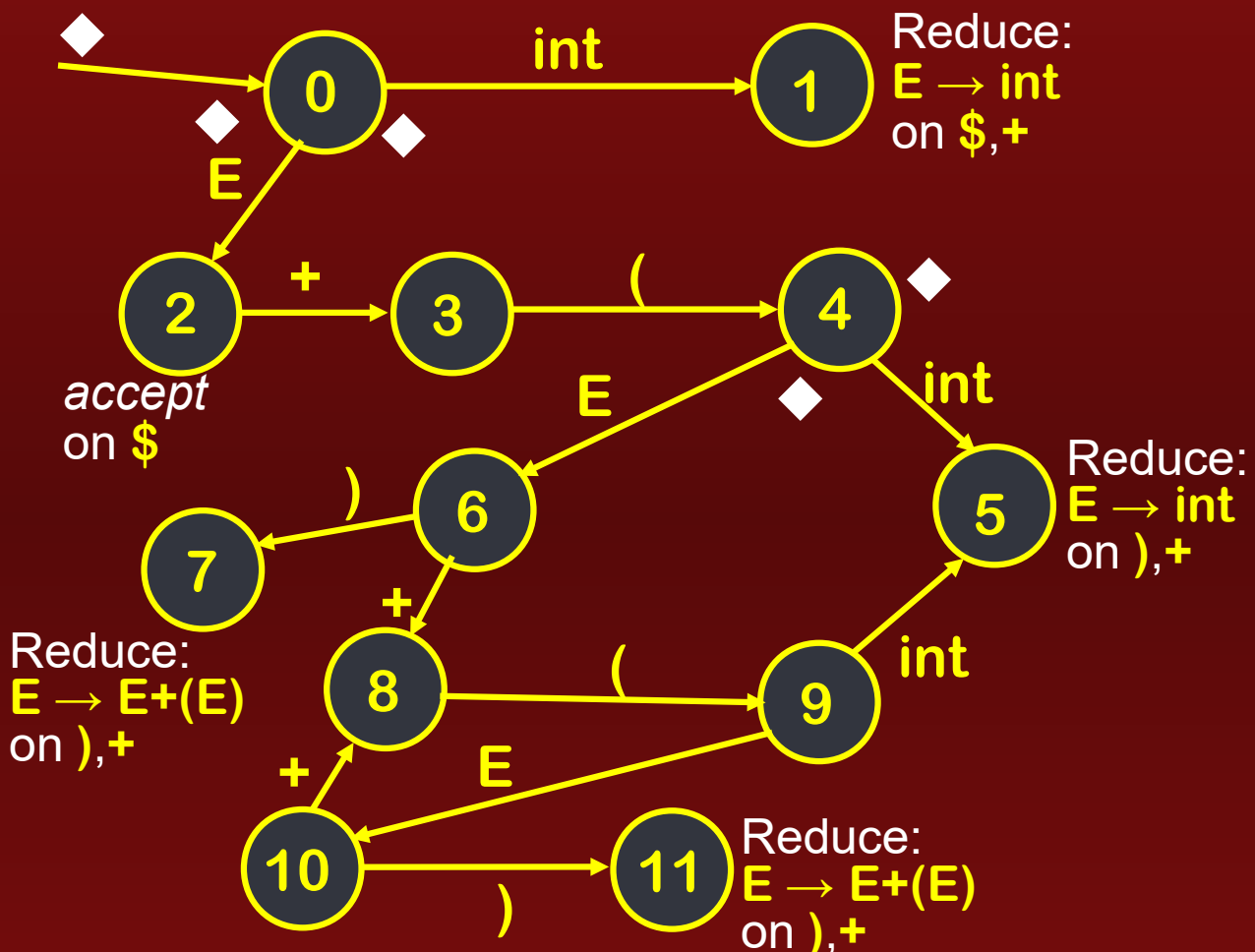
- The LR(1) construction algorithm builds a handle-recognizing DFA
- The parsing algorithm uses this DFA to recognize handles and potential handles on the parse stack

17

LR(1) Parsers

- The LR(1) construction algorithm builds a handle-recognizing DFA
- The parsing algorithm **uses** this **DFA** to **recognize handles** and potential handles on the parse **stack**

18



19

Parsing DFA

▶ int + (int) + (int) \$	shift 1
int ▶ + (int) + (int) \$	reduce E → int
E ▶ + (int) + (int) \$	shift 3 times
E + (int ▶) + (int) \$	reduce E → int
E + (E ▶) + (int) \$	shift
E + (E) ▶ + (int) \$	red E → E+(E)

20

Parsing DFA

- How can the parser **remember** the **state** the DFA goes into on a symbol?

21

Parsing DFA

- Store the DFA **state** in the **stack** along with the symbol
- Initial entry in the **stack** will be '**<dummy,0>**'.

22

▪ 0 ► **int + (int)+(int) \$**

▪ 0 **int** 1 ► **+(int)+(int)\$**

▪ 0 **E** 2 ► **+(int)+(int)\$**

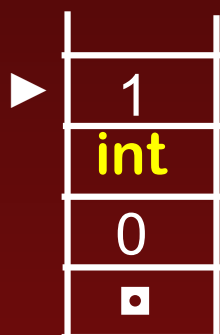
... ..

shift 1

reduce **E** → **int**

shift 3 times

... ..



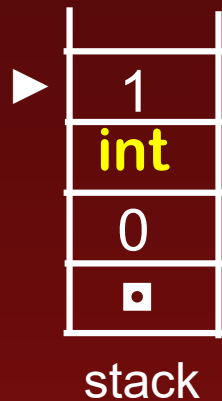
stack

▪ is dummy symbol

23

□ 0 ► **int** + (int)+(int) \$
 □ 0 **int** 1 ► **+**(int)+(int)\$
 □ 0 **E** 2 ► **+**(int)+(int)\$

shift 1
 reduce **E** → **int**
 shift 3 times



24

Parsing DFA

□ 0 ► **int** + (int)+(int) \$
 □ 0 **int** 1 ► **+**(int)+(int)\$
 → □ 0 **E** 2 ► **+**(int)+(int)\$

shift 1
 reduce **E** → **int**
 shift 3 times

How is next state decided upon a reduce?

25

Representing DFA

- Parsers represent **DFA** as a 2D table
- Rows correspond to DFA states
- Columns correspond to terminals and non-terminals

26

Representing DFA

- Parsers represent **DFA** as a 2D table
- **Rows** correspond to DFA states
- Columns correspond to terminals and non-terminals

27

Representing DFA

- Parsers represent DFA as a 2D table
- Rows correspond to DFA states
- Columns correspond to terminals and non-terminals

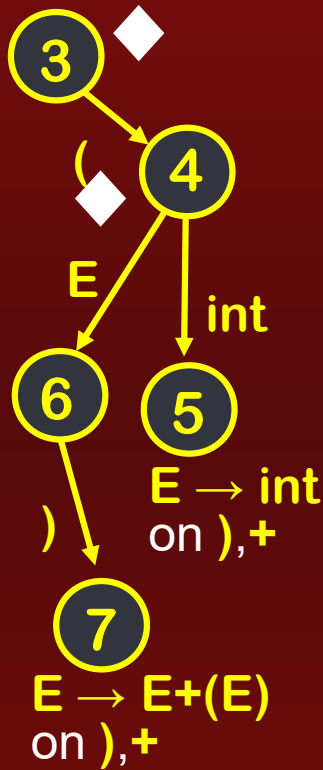
28

Representing DFA

columns are split into:

- those for terminals:
action table
- those for non-terminals:
goto table

29



	action					goto
state	int	+	(	)	\$	E
...						
3			s4			
4	s5					g6
5		r2		r2		
6	s8		s7			
7		r1			r1	
...						

30

Parsing DFA

- 0 ► int + (int)+(int) \$
- 0 int 1 ► +(int)+(int)\$
- 0 E 2 ► +(int)+(int)\$

... ..

shift 1

reduce $E \rightarrow int$

shift 3 times

... ..

The **state** comes from the **goto** table

31

Building LR(1) Tables

- To **construct** *Action* and *Goto* tables, the LR(1) parser generator builds a **model** of handle-recognizing **DFA**
- The model is used to fill in the tables

32

Building LR(1) Tables

- To construct *Action* and *Goto* tables, the LR(1) parser generator builds a model of handle-recognizing DFA
- The **model** is used to **fill** in the tables

33

LR(1) Items

- The LR(1)-Table construction needs a concrete representation for the handles and their associated lookahead symbols.
- We call this representation an LR(1) item.

34

LR(1) Items

- The LR(1)-Table construction needs a concrete representation for the handles and their associated lookahead symbols.
- We call this representation an LR(1) item.

35

Compiler Construction

Sohail Aslam

Lecture 24

LR(1) Items

- An LR(1) item is a pair:

$$[X \rightarrow \alpha \bullet \beta, a]$$

- $X \rightarrow \alpha \beta$ is a production
- $a \in T$ is lookahead symbol

Building LR(1) Tables

- The model uses a set of LR(1) items to represent each parser state
- The model is called the *canonical collection of set of LR(1) items*

3

Building LR(1) Tables

- The model uses a set of LR(1) items to represent each parser state
- The model is called the *canonical collection (CC) of set of LR(1) items*

4

Canonical Collection

- Each set in CC represents a state in the eventual parser DFA
- The construction of CC begins by building a model of parser's initial state

5

Canonical Collection

- Each set in CC represents a state in the eventual parser DFA
- The construction of CC begins by building a model of parser's initial state

6

Canonical Collection

- The initial state consists of the set of LR(1) items that represent the parser's initial state, along with any items that must also hold in the initial state.

7

Canonical Collection

- To simplify the task of building this initial state, the construction requires that the grammar have a unique goal symbol

8

Convention

- Add a new start symbol **S** to grammar and a production

$$\mathbf{S} \rightarrow \mathbf{E}$$

- Augmented grammar

$$\mathbf{S} \rightarrow \mathbf{E}$$

$$\mathbf{E} \rightarrow \mathbf{E} + (\mathbf{E}) \mid \underline{\text{int}}$$

9

Convention

- Add a new start symbol **S** to grammar and a production

$$\mathbf{S} \rightarrow \mathbf{E}$$

- Augmented grammar

$$\mathbf{S} \rightarrow \mathbf{E}$$

$$\mathbf{E} \rightarrow \mathbf{E} + (\mathbf{E}) \mid \underline{\text{int}}$$

10

The *Closure* Procedure

- The item $[S \rightarrow \bullet E, \$]$ describes the parser's initial state.
- It represents a configuration in which recognizing S followed by $\$$ would be a valid parse

11

The *Closure* Procedure

- The item $[S \rightarrow \bullet E, \$]$ describes the parser's initial state.
- It represents a configuration in which recognizing S followed by $\$$ would be a valid parse

12

The *Closure* Procedure

- This item, i.e., $[S \rightarrow \bullet E, \$]$ becomes the core of the first state in *CC*, labelled I_0 .

13

The *Closure* Procedure

- If the *grammar* has several *distinct* productions for the start symbol, *each* of them *generates* an *item* in this initial core of I_0
- The procedure *closure* does this.

14

The *Closure* Procedure

- If the grammar has several distinct productions for the start symbol, each of them generates an item in this initial core of I_0
- The procedure *closure* does this.

15

$\text{closure}(s) =$

repeat

for each $[X \rightarrow \alpha \bullet Y \beta, a] \in s$

for each production $Y \rightarrow \gamma$

for each $b \in \text{FIRST}(\beta a)$

$s \leftarrow s \cup [Y \rightarrow \bullet \gamma, b]$

until s is unchanged

16

closure(**s**) =

repeat

for each $[X \rightarrow \alpha \bullet Y \beta, a] \in s$

for each production $Y \rightarrow \gamma$

for each $b \in \text{FIRST}(\beta a)$

$s \leftarrow s \cup [Y \rightarrow \bullet \gamma, b]$

until **s** is unchanged

17

◆ $I_0 = \text{closure}(\{[S \rightarrow \bullet E, \$]\})$

◆ $s = \{[S \rightarrow \bullet E, \$]\}$

◆ $[X \rightarrow \alpha \bullet Y \beta, a] \Leftrightarrow [S \rightarrow \bullet E, \$]$

$X = S, \alpha = \varepsilon, Y = E$

$\beta = \varepsilon, a = \$$

◆ $Y \rightarrow \gamma \square \Leftrightarrow E \rightarrow E + (E)$

$\Leftrightarrow E \rightarrow \text{int } \square$

18

closure(**s**) =

repeat

for each $[X \rightarrow \alpha \bullet Y \beta, a] \in s$

for each production $Y \rightarrow \gamma$

for each $b \in \text{FIRST}(\beta a)$

$s \leftarrow s \cup [Y \rightarrow \bullet \gamma, b]$

until **s** is unchanged

19

$\text{FIRST}(\beta a) = \text{FIRST}(\$) = \$$

$s = \{ [S \rightarrow \bullet E, \$] \} \cup$
 $\{ [E \rightarrow \bullet E + (E), \$] \} \cup$
 $\{ [E \rightarrow \bullet \text{int}, \$] \}$

20

$$\text{FIRST}(\beta a) = \text{FIRST}(\$) = \$$$

$$s = \{ [S \rightarrow \bullet E, \$] \} \cup \\ \{ [E \rightarrow \bullet E + (E), \$] \} \cup \\ \{ [E \rightarrow \bullet \text{int}, \$] \}$$

21

$$\text{FIRST}(\beta a) = \text{FIRST}(\$) = \$$$

$$s = \{ [S \rightarrow \bullet E, \$] \} \cup \\ \{ [E \rightarrow \bullet E + (E), \$] \} \cup \\ \{ [E \rightarrow \bullet \text{int}, \$] \}$$

22

$$\text{FIRST}(\beta a) = \text{FIRST}(\$) = \$$$

$$s = \{ [S \rightarrow \bullet E, \$] \} \cup \\ \{ [E \rightarrow \bullet E + (E), \$] \} \cup \\ \{ [E \rightarrow \bullet \text{int}, \$] \}$$

23

$$s = \{ [S \rightarrow \bullet E, \$] , \\ [E \rightarrow \bullet E + (E), \$] , \\ [E \rightarrow \bullet \text{int}, \$] \}$$

24

closure(**s**) =

repeat

for each $[X \rightarrow \alpha \bullet Y \beta, a] \in s$

for each production $Y \rightarrow \gamma$

for each $b \in \text{FIRST}(\beta a)$

$s \leftarrow s \cup [Y \rightarrow \bullet \gamma, b]$

until **s** is unchanged

25

◆ $[S \rightarrow \bullet E, \$]$ already processed

◆ $[X \rightarrow \alpha \bullet Y \beta, a] \Leftrightarrow [E \rightarrow \bullet E + (E), \$]$

◆ $X = E, \alpha = \varepsilon, Y = E$
 $\beta = +(E), a = \$$

◆ $Y \rightarrow \gamma \square \Leftrightarrow E \rightarrow E + (E)$
 $\Leftrightarrow E \rightarrow \text{int } \square$

26

$$\text{FIRST}(\beta a) =$$

$$\text{FIRST}(+(E)\$) = +$$

$$s = s \cup \{ [E \rightarrow \bullet E+(E), +] \}$$

$$\cup \{ [E \rightarrow \bullet \text{int}, +] \}$$

27

$$\text{FIRST}(\beta a) =$$

$$\text{FIRST}(+(E)\$) = +$$

$$s = s \cup \{ [E \rightarrow \bullet E+(E), +] \}$$

$$\cup \{ [E \rightarrow \bullet \text{int}, +] \}$$

28

Compiler Construction

Sohail Aslam

Lecture 25

Sets of LR(1) Items

$$I_0 = \{$$

- ◆ $[S \rightarrow \bullet E, \$],$
- $[E \rightarrow \bullet E + (E), \$ / +],$
- $[E \rightarrow \bullet \text{int}, \$ / +]$

$$\}$$

The *Closure* Procedure

Rationale:

- if $[A \rightarrow \beta \bullet C \delta, a] \in s$, then one potential **completion** for the **left context** is to find a string that **reduces** to **C**, followed by **δa**

3

The *Closure* Procedure

- This completion should cause a **reduction** to **A**, since it fills out the production's **right-hand** side (**$C\delta$**), and follows it with a valid lookahead symbol

4

The *Closure* Procedure

- For a production $C \rightarrow \gamma$, *closure* must insert ' $\bullet$ ' before γ and add appropriate *lookahead* symbols – all terminals that can appear as the initial symbol in δa

5

The *Closure* Procedure

- This includes every terminal in $FIRST(\delta)$.
If $\epsilon \in FIRST(\delta)$, it also includes a , thus $FIRST(\delta a)$ in the algorithm

6

The *goto* Procedure

- The **second** critical step in the construction is to **derive other parser states** from I_0

7

The *goto* Procedure

- To accomplish this, we **compute**, for each **state** I_i and each grammar **symbol** y , the **state** that would **arise** if the **parser recognized** an y while in state I_i

8

The *goto* Procedure

- A state s that contains $[X \rightarrow \alpha \bullet y \beta, b]$ has a transition (*goto*) labeled y to the state that contains the items $goto(s, y)$
 - y can be *terminal* or a *non-terminal*

9

$goto(s, y)$

$m \leftarrow \{ \}$

for each item $[X \rightarrow \alpha \bullet y \beta, b] \in s$

$m \leftarrow m \cup \{[X \rightarrow \alpha y \bullet \beta, b]\}$

return $closure(m)$

Finite Automaton of Items

- The LR(1) items are used as the states of a finite automaton that maintains information about the parsing stack and progress of a shift-reduce parser

11

Finite Automaton of Items

- This will start out as a nondeterministic finite automaton (NFA)
- The DFA can be constructed from this NFA using the subset construction, similar to one we used for lexical analysis

12

Finite Automaton of Items

- This will start out as a nondeterministic finite automaton (NFA)
- The **DFA** can be constructed from this NFA using the **subset construction**, similar to one we used for lexical analysis

13

Finite Automaton of Items

- What are the **transitions** of the **NFA** of **LR(0)** items?
- Consider the item $A \rightarrow \alpha \bullet \gamma$
- Suppose γ begins with symbol X which may be a terminal (token) or non-terminal

14

Finite Automaton of Items

- What are the transitions of the NFA of LR(0) items?
- Consider the item $A \rightarrow \alpha \bullet \gamma$
- Suppose γ begins with symbol X which may be a terminal (token) or non-terminal

15

Finite Automaton of Items

- What are the transitions of the NFA of LR(0) items?
- Consider the item $A \rightarrow \alpha \bullet \gamma$
- Suppose γ begins with symbol X which may be a terminal (token) or non-terminal

16

Finite Automaton of Items

- The item can be written as $A \rightarrow \alpha \bullet X \eta$
- Then there is a transition on symbol X for state represented by item $A \rightarrow \alpha \bullet X \eta$ to state represented by item $A \rightarrow \alpha X \bullet \eta$

17

Finite Automaton of Items

- The item can be written as $A \rightarrow \alpha \bullet X \eta$
- Then there is a transition on symbol X for state represented by item $A \rightarrow \alpha \bullet X \eta$ to state represented by item $A \rightarrow \alpha X \bullet \eta$

18

Finite Automaton of Items



19

Finite Automaton of Items



If X is a **terminal**, then this transition corresponds to a **shift** of X from **input** to top of **parse stack**

20

Finite Automaton of Items



If X is a **non-terminal**, then the interpretation of this transition is more complex because non-terminals **do not** appear in **input**

21

Finite Automaton of Items



In fact, such a transition will correspond to **pushing** of X onto the **stack** during the parse

22

Finite Automaton of Items



But this can only occur during a **reduction** by the production $X \rightarrow \beta$

23

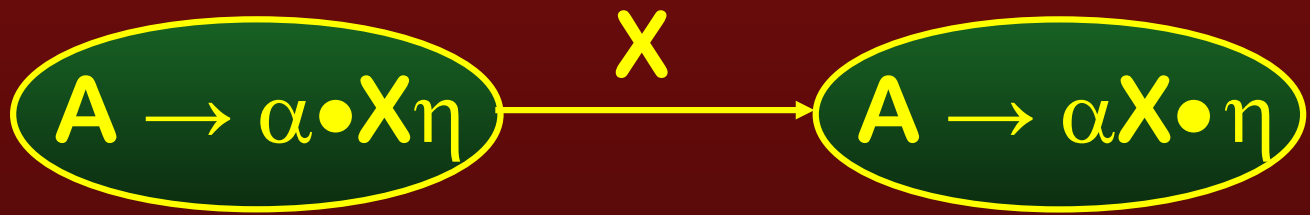
Finite Automaton of Items



Such a reduction must be **preceded** by **recognition** of a β

24

Finite Automaton of Items



The **state** given by $X \rightarrow \bullet \beta$ represents the **beginning** of this process (dot indicates we are about to recognize β)

25

Compiler Construction

Sohail Aslam

Lecture 26

Finite Automaton of Items

Then for every item $A \rightarrow \alpha \bullet X \eta$ we must **add** an ε -transition for every production $X \rightarrow \beta$



2

Finite Automaton of Items

- The **initial** DFA state I_0 we computed is the ε -closure of the set consisting of item

$$S \rightarrow \bullet E$$

3

Finite Automaton of Items

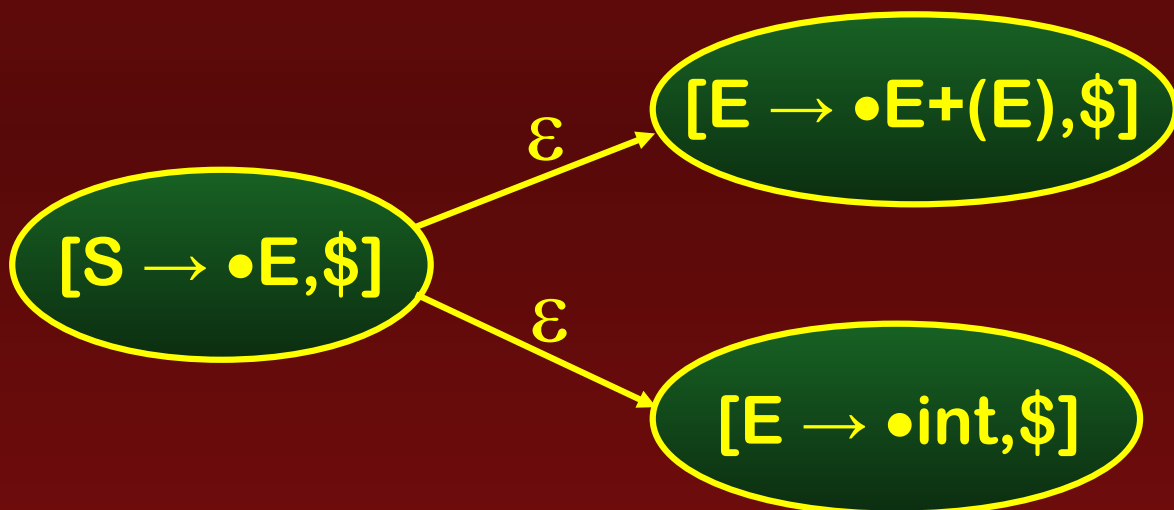
- Recall the stage in the closure

$$s = \{ [S \rightarrow \bullet E, \$], \\ [E \rightarrow \bullet E + (E), \$], \\ [E \rightarrow \bullet \text{int}, \$] \}$$

4

Finite Automaton of Items

NFA states



5

Algorithm:

Construction of collection of canonical sets of LR(1) items.

Input:

An augmented grammar G'

Output:

Collection of canonical (CC) sets of LR(1)

6

Algorithm:

Construction of collection of canonical sets of LR(1) items.

Input:

An augmented grammar G'

Output:

Collection of canonical (CC) sets of LR(1)

7

Algorithm:

Construction of collection of canonical sets of LR(1) items.

Input:

An augmented grammar **G'**

Output:

Collection of canonical (**CC**) sets of LR(1)

8

CC(**G'**)

$I_0 \leftarrow \{\text{closure}([S' \rightarrow \bullet S, \$])\}$

CC $\leftarrow \{I_0\}$

repeat

for each unmarked set $I_j \in \mathbf{CC}$

mark I_j as processed

*for each **X** following $\bullet$ in an item in I_j*

$I_k \leftarrow \text{goto}(I_j, \mathbf{X})$

if $I_k \notin \mathbf{CC}$ then

$\mathbf{CC} \leftarrow \mathbf{CC} \cup I_k$

*record transition from I_j to I_k on **X***

*until **CC** is not changing*

9

Sets of LR(1) Items

Augmented grammar G'

$S \rightarrow E$

$E \rightarrow E + (E) \mid \underline{\text{int}}$

10

Sets of LR(1) Items

- We now compute $\text{goto}(I_0, X)$ for various values of X
- X can be E , int , $+$, $($ and $)$

11

goto(*s*, **y**)

$m \leftarrow \{ \}$

for each item $[X \rightarrow \alpha \bullet y \beta, b] \in s$

$m \leftarrow m \cup \{[X \rightarrow \alpha y \bullet \beta, b]\}$

return *closure*(*m*)

12

$I_1 = \text{goto}(I_0, \text{int})$

$m \leftarrow \{ \}$

for each $[X \rightarrow \alpha \bullet \text{int} \beta, b] \in I_0$

$\Leftrightarrow [E \rightarrow \bullet \text{int}, \$ / +] \in I_0$

$m = m \cup \{[E \rightarrow \text{int} \bullet, \$ / +]\}$

return *closure*($\{[E \rightarrow \text{int} \bullet, \$ / +]\}$)

13

$\downarrow$
 $\text{closure}(\{[E \rightarrow \text{int}\bullet, \$/+]\})$

No additional closure is possible since the dot is at the right end of the production

14

- Thus $I_1 = \{[E \rightarrow \text{int}\bullet, \$/+]\}$
- And we have the transition from I_0 to I_1 on **int**

15

- Thus $I_1 = \{[E \rightarrow \text{int}\bullet, \$/+]\}$
- And we have the transition from I_0 to I_1 on **int**



16

$I_2 = \text{goto}(I_0, E)$

$m \leftarrow \{\}$

for each $[X \rightarrow \alpha\bullet E\beta, b] \in I_0$

$\Leftrightarrow [S \rightarrow \bullet E, \$]$

$\Leftrightarrow [E \rightarrow \bullet E+(E), \$/+]$

17

$m = m \cup \{[S \rightarrow E\bullet, \$]\} \cup$
 $\{[E \rightarrow E\bullet+(E), \$/+]\}$
return closure(m)

18

$I_2 = \{[S \rightarrow E\bullet, \$],$
 $[E \rightarrow E\bullet+(E), \$/+]\}$

- No further closure for the first item because $\bullet$ is at the **end**
- In the second item, a terminal $+$ appears after $\bullet$ so no further closure

19

$$I_2 = \{[S \rightarrow E\bullet, \$], \\ [E \rightarrow E\bullet + (E), \$/+]\}$$


- No further closure for the first item because $\bullet$ is at the end
- In the second item, a **terminal $+$** appears after $\bullet$ so no further closure

20

- $I_3 = \text{goto}(I_2, +) =$
 $\{[E \rightarrow E + \bullet (E), \$/+]\}$



- $I_4 = \text{goto}(I_3, ($
 $= \{ [E \rightarrow E + (\bullet E), \$/+]$
 $[E \rightarrow \bullet E + (E),)/+]$
 $[E \rightarrow \bullet \text{int},)/+]\}$

21

- $I_3 = \text{goto}(I_2, +) =$
 $\{[E \rightarrow E + \bullet (E), \$/+]\}$

- $I_4 = \text{goto}(I_3, ()$
 $= \{ [E \rightarrow E + (\bullet E), \$/+]$
 $[E \rightarrow \bullet E + (E),)/+]$
 $[E \rightarrow \bullet \underline{\text{int}},)/+]\}$

22

- $I_5 = \text{goto}(I_4, \text{int}) =$
 $\{ [E \rightarrow \text{int} \bullet,)/+]\}$

- $I_6 = \text{goto}(I_4, E) = \{$
 $[E \rightarrow E + (E \bullet), \$/+]$
 $[E \rightarrow E \bullet + (E),)/+]\}$

- and so on

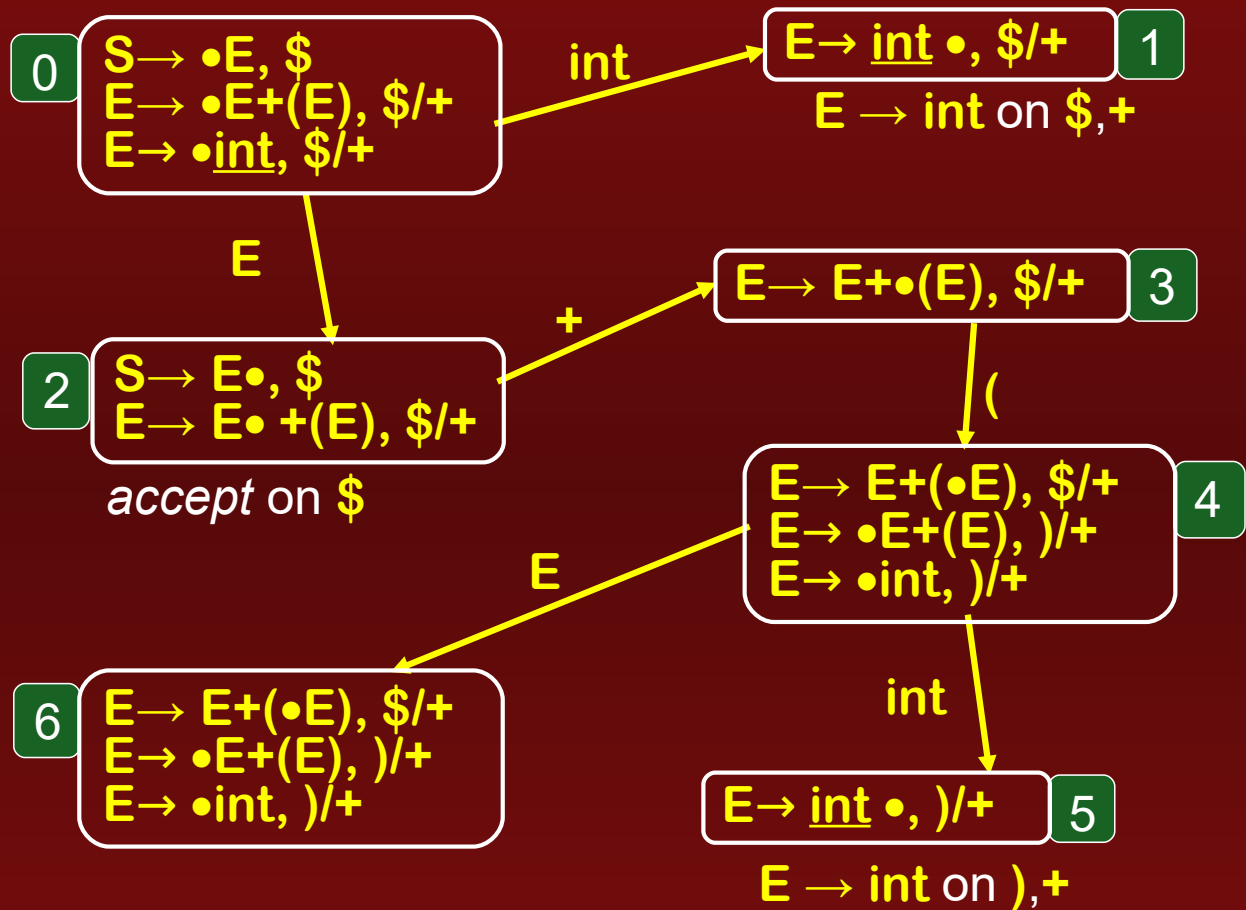
23

- $I_5 = \text{goto}(I_4, \text{int}) = \{ [E \rightarrow \text{int} \bullet,)/+] \}$
- $I_6 = \text{goto}(I_4, E) = \{ [E \rightarrow E + (E \bullet), \$/+]$
 $[E \rightarrow E \bullet + (E),)/+] \}$
- and so on

24

- $I_5 = \text{goto}(I_4, \text{int}) = \{ [E \rightarrow \text{int} \bullet,)/+] \}$
- $I_6 = \text{goto}(I_4, E) = \{ [E \rightarrow E + (E \bullet), \$/+]$
 $[E \rightarrow E \bullet + (E),)/+] \}$
- and so on

25



26

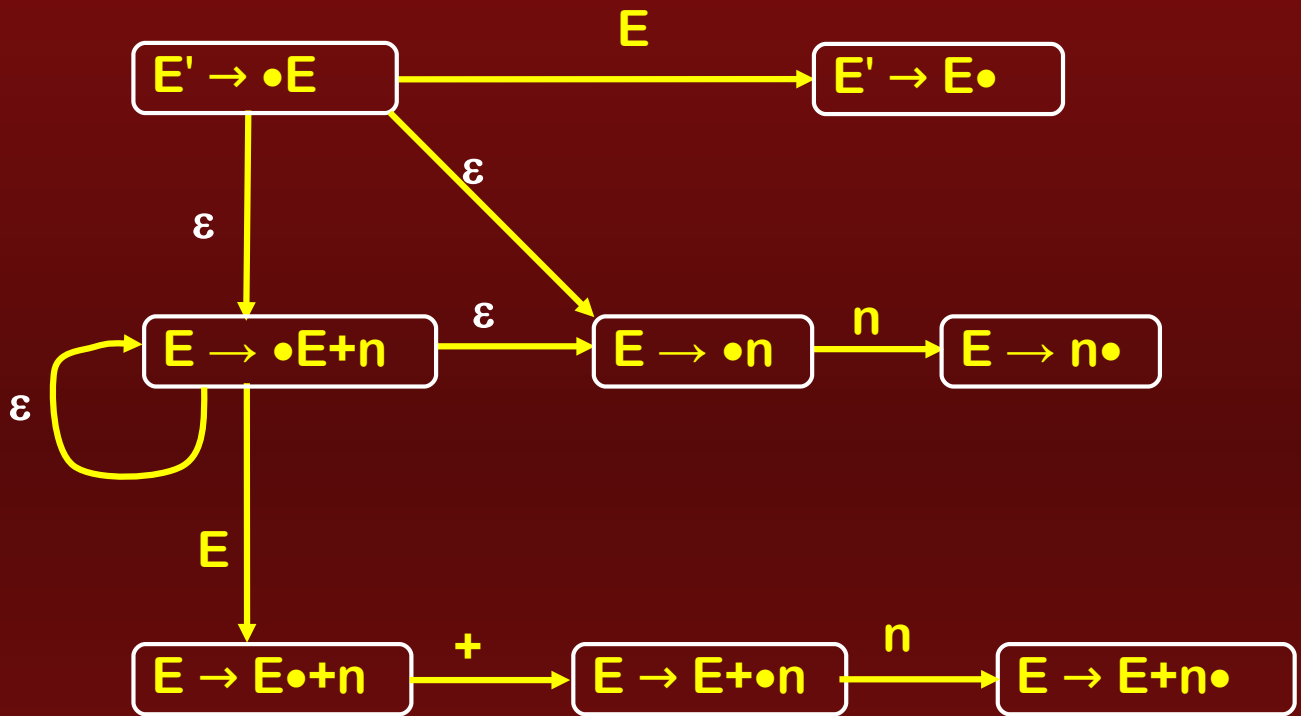
NFA of LR(0) Items

Consider the augmented grammar

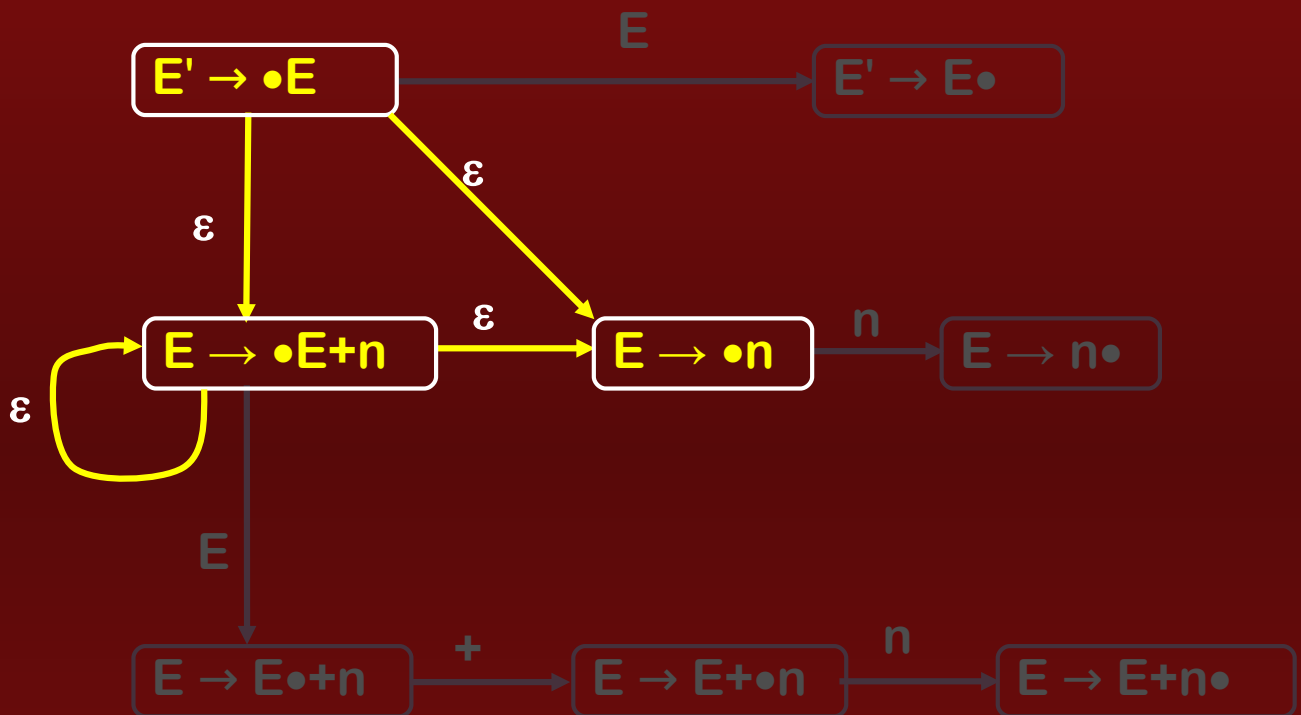
$$E' \rightarrow E$$

$$E \rightarrow E + n \mid n$$

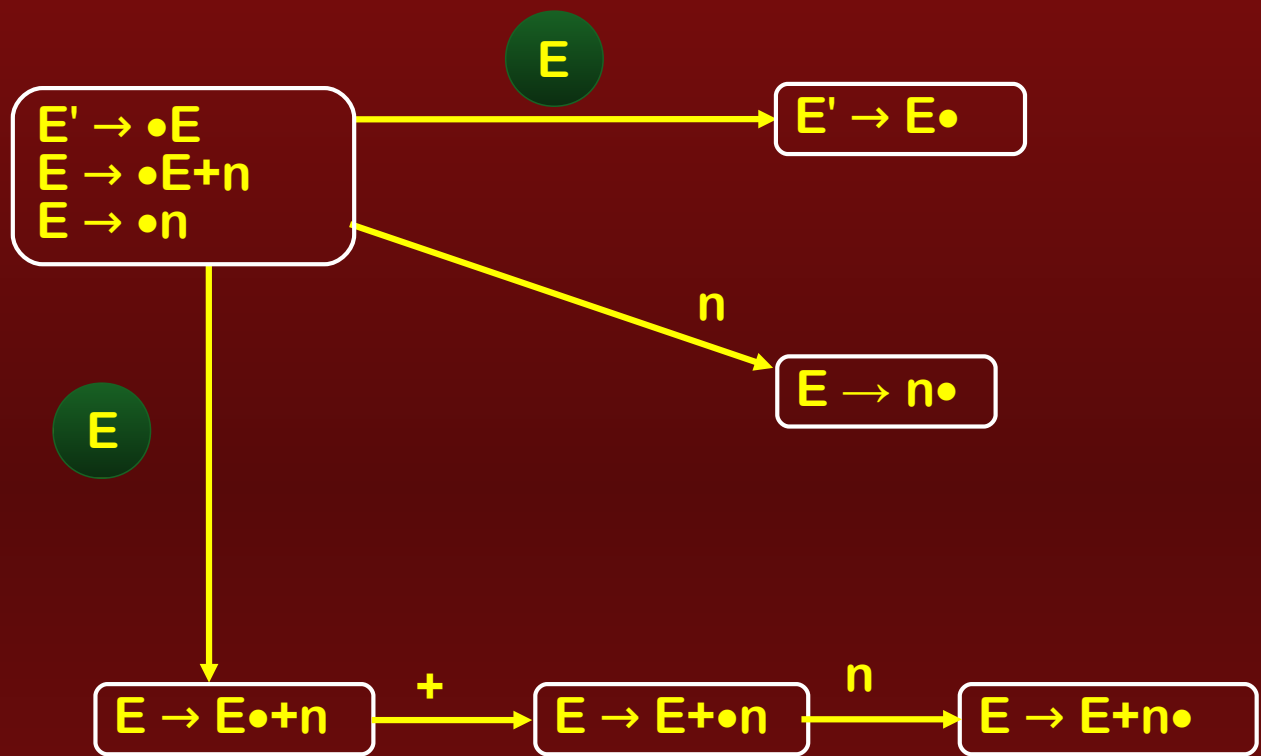
27



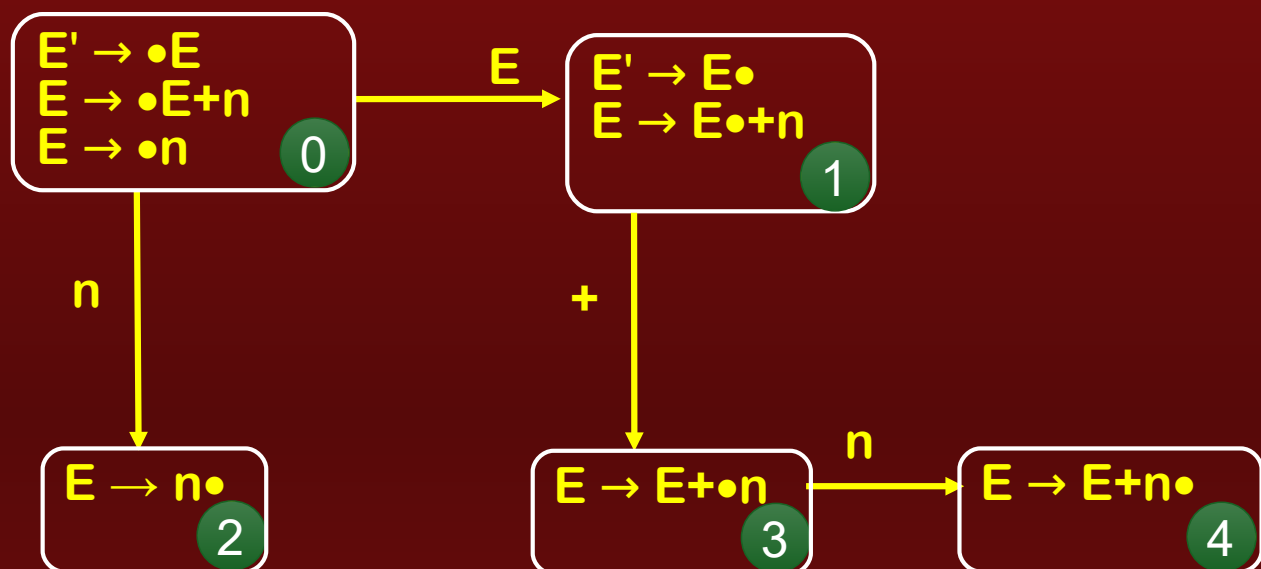
28



29



30



31

LR(1) Transitions

Given an LR(1) item

$$[A \rightarrow \alpha \bullet B \gamma, a]$$

where **B** is a non-terminal, there are **ϵ -transitions** to items

$[B \rightarrow \bullet \beta, b]$ for every production **$B \rightarrow \beta$** and for every token **b** in **FIRST(γa)**

32

LR Table Construction

- Construct **CC** = $\{I_0, I_1, I_2, \dots, I_n\}$, for **G'**
- State *i* of the parser is constructed from the set I_i
- The parsing actions for state *i* are determined as follows:

33

LR Table Construction

- Construct $CC = \{I_0, I_1, I_2, \dots, I_n\}$, for G'
- **State i** of the parser is constructed from the set I_i
- The parsing actions for state i are determined as follows:

34

LR Table Construction

- Construct $CC = \{I_0, I_1, I_2, \dots, I_n\}$, for G'
- **State i** of the parser is constructed from the set I_i
- The parsing **actions** for **state i** are determined as follows:

35

Compiler Construction

Sohail Aslam

Lecture 27

for each item $I_i \in CC$

- *if $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $\text{goto}(I_i, a) == I_j$ then*
 Action[i, a] $\leftarrow$ “shift j”
- *else if $[A \rightarrow \alpha \bullet, a] \in I_i$ and $A \neq S'$ then*
 Action[i, a] $\leftarrow$ “reduce $A \rightarrow \alpha$ ”

- else if $[S' \rightarrow S\bullet, \$] \in I_i$ then

 Action[i, a] $\leftarrow$ “accept”

end for

// the goto table

for each non-terminal $A \in G$

 if goto(I_i, A) = I_j then

 Goto[i, A] $\leftarrow j$

3

LR Table Construction

- The initial state is the one that contains the item

$[S' \rightarrow \bullet S, \$]$

- All remaining entries are marked “error”

4

LR Table Construction

- The initial state is the one that contains the item $[S' \rightarrow \bullet S, \$]$
- All remaining entries are marked “error”

5

Table Construction Example

1. $S' \rightarrow E$
2. $E \rightarrow T - E$
3. $E \rightarrow T$
4. $T \rightarrow F \times T$
5. $T \rightarrow F$
6. $F \rightarrow id$

6

Symbol	FIRST
S'	{ id }
E	{ id }
T	{ id }
F	{ id }
id	{ id }
×	{ × }
–	{ – }

7

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

{ $[S' \rightarrow \bullet E, \$]$,
 $[E \rightarrow \bullet T - E, \$]$, $[E \rightarrow \bullet T, \$]$,
 $[T \rightarrow \bullet F \times T, \$]$, $[T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -]$, $[T \rightarrow \bullet F, -]$,
 $[F \rightarrow \bullet id, \$]$, $[F \rightarrow \bullet id, -]$,
 $[F \rightarrow \bullet id, \times]$
 }

8

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

$\{ [S' \rightarrow \bullet E, \$],$
[E $\rightarrow \bullet T - E, \$$], [E $\rightarrow \bullet T, \$$],
[T $\rightarrow \bullet F \times T, \$$], [T $\rightarrow \bullet F, \$$]
[T $\rightarrow \bullet F \times T, -$], [T $\rightarrow \bullet F, -$],
[F $\rightarrow \bullet \text{id}, \$$], [F $\rightarrow \bullet \text{id}, -$],
[F $\rightarrow \bullet \text{id}, \times$]
 $\}$

9

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

$\{ [S' \rightarrow \bullet E, \$],$
[E $\rightarrow \bullet T - E, \$$], [E $\rightarrow \bullet T, \$$],
[T $\rightarrow \bullet F \times T, \$$], [T $\rightarrow \bullet F, \$$]
[T $\rightarrow \bullet F \times T, -$], [T $\rightarrow \bullet F, -$],
[F $\rightarrow \bullet \text{id}, \$$], [F $\rightarrow \bullet \text{id}, -$],
[F $\rightarrow \bullet \text{id}, \times$]
 $\}$

10

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

{ $[S' \rightarrow \bullet E, \$]$,
 $[E \rightarrow \bullet T - E, \$]$, $[E \rightarrow \bullet T, \$]$,
 $[T \rightarrow \bullet F \times T, \$]$, $[T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -]$, $[T \rightarrow \bullet F, -]$,
 $[F \rightarrow \bullet \text{id}, \$]$, $[F \rightarrow \bullet \text{id}, -]$,
 $[F \rightarrow \bullet \text{id}, \times]$
}

11

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

{ $[S' \rightarrow \bullet E, \$]$,
 $[E \rightarrow \bullet T - E, \$]$, $[E \rightarrow \bullet T, \$]$,
 $[T \rightarrow \bullet F \times T, \$]$, $[T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -]$, $[T \rightarrow \bullet F, -]$,
 $[F \rightarrow \bullet \text{id}, \$]$, $[F \rightarrow \bullet \text{id}, -]$,
 $[F \rightarrow \bullet \text{id}, \times]$
}

12

$$I_1 = \{\text{goto}(I_0, E)\} \\ \{[S' \rightarrow E\bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_0, T)\} \\ \{[E \rightarrow T\bullet - E, \$], [E \rightarrow T\bullet, \$]\}$$

13

$$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\} \\ \{ [S' \rightarrow \bullet E, \$], \\ [E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$], \\ [T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F, \$] \\ [T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, -], \\ [F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -], \\ [F \rightarrow \bullet \text{id}, \times] \\ \}$$

14

$$I_1 = \{\text{goto}(I_0, E)\}$$

$$\{[S' \rightarrow E\bullet, \$]\}$$


$$I_2 = \{\text{goto}(I_0, T)\}$$

$$\{[E \rightarrow T\bullet - E, \$], [E \rightarrow T\bullet, \$]\}$$

15

$$I_1 = \{\text{goto}(I_0, E)\}$$

$$\{[S' \rightarrow E\bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_0, T)\}$$

$$\{[E \rightarrow T\bullet - E, \$], [E \rightarrow T\bullet, \$]\}$$

16

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

$\{ [S' \rightarrow \bullet E, \$],$
 $[E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$],$
 $[T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, -],$
 $[F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -],$
 $[F \rightarrow \bullet \text{id}, \times]$
 $\}$

17

$I_1 = \{\text{goto}(I_0, E)\}$
 $\{[S' \rightarrow E \bullet, \$]\}$

$I_2 = \{\text{goto}(I_0, T)\}$
 $\{[E \rightarrow T \bullet - E, \$], [E \rightarrow T \bullet, \$]\}$

18

$I_3 = \{\text{goto}(I_0, F)\}$

$\{ [T \rightarrow F \bullet \times T, \$],$
 $[T \rightarrow F \bullet, \$],$
 $[T \rightarrow F \bullet \times T, -],$
 $[T \rightarrow F \bullet, -]$
 $\}$

19

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

$\{ [S' \rightarrow \bullet E, \$],$
 $[E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$],$
 $[T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, -],$
 $[F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -],$
 $[F \rightarrow \bullet \text{id}, \times]$
 $\}$

20

$$I_3 = \{\text{goto}(I_0, F)\}$$

$$\{ [T \rightarrow F \bullet \times T, \$],$$

$$[T \rightarrow F \bullet, \$],$$

$$[T \rightarrow F \bullet \times T, -],$$

$$[T \rightarrow F \bullet, -]$$

$$\}$$

21

$$I_4 = \{\text{goto}(I_0, id)\}$$

$$\{ [F \rightarrow id \bullet, \$],$$

$$[F \rightarrow id \bullet, -],$$

$$[F \rightarrow id \bullet, \times]$$

$$\}$$

22

$I_0 = \{\text{closure}([S' \rightarrow \bullet E, \$])\}$

$\{$ $[S' \rightarrow \bullet E, \$]$,
 $[E \rightarrow \bullet T - E, \$]$, $[E \rightarrow \bullet T, \$]$,
 $[T \rightarrow \bullet F \times T, \$]$, $[T \rightarrow \bullet F, \$]$
 $[T \rightarrow \bullet F \times T, -]$, $[T \rightarrow \bullet F, -]$,
 $[F \rightarrow \bullet id, \$]$, $[F \rightarrow \bullet id, -]$,
 $[F \rightarrow \bullet id, \times]$
 $\}$

23

$I_4 = \{\text{goto}(I_0, id)\}$

$\{$ $[F \rightarrow id \bullet, \$]$,
 $[F \rightarrow id \bullet, -]$,
 $[F \rightarrow id \bullet, \times]$
 $\}$

24

$I_5 = \{\text{goto}(I_2, -)\}$

$\{[E \rightarrow T- \bullet E, \$],$

$[E \rightarrow \bullet T-E, \$],$

$[E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$],$

$[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$],$

$[T \rightarrow \bullet F, -], [F \rightarrow \bullet \text{id}, \$],$

$[F \rightarrow \bullet \text{id}, -], [F \rightarrow \bullet \text{id}, \times] \}$

25

$I_5 = \{\text{goto}(I_2, -)\}$

$\{[E \rightarrow T- \bullet E, \$],$

$[E \rightarrow \bullet T-E, \$],$

$[E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$],$

$[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$],$

$[T \rightarrow \bullet F, -], [F \rightarrow \bullet \text{id}, \$],$

$[F \rightarrow \bullet \text{id}, -], [F \rightarrow \bullet \text{id}, \times] \}$

26

$I_5 = \{\text{goto}(I_2, -)\}$

$\{[E \rightarrow T- \bullet E, \$],$

$[E \rightarrow \bullet T-E, \$],$

$[E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$],$

$[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$],$

$[T \rightarrow \bullet F, -], [F \rightarrow \bullet \text{id}, \$],$

$[F \rightarrow \bullet \text{id}, -], [F \rightarrow \bullet \text{id}, \times] \}$

27

$I_5 = \{\text{goto}(I_2, -)\}$

$\{[E \rightarrow T- \bullet E, \$],$

$[E \rightarrow \bullet T-E, \$],$

$[E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$],$

$[T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$],$

$[T \rightarrow \bullet F, -], [F \rightarrow \bullet \text{id}, \$],$

$[F \rightarrow \bullet \text{id}, -], [F \rightarrow \bullet \text{id}, \times] \}$

28

$$I_6 = \{\text{goto}(I_3, \times)\}$$

$\{[T \rightarrow F \times \bullet T, \$], [T \rightarrow F \times \bullet T, -],$

$[T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -],$

$[T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -],$

$[F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -],$

$[F \rightarrow \bullet \text{id}, \times] \}$

29

$$I_6 = \{\text{goto}(I_3, \times)\}$$

$\{[T \rightarrow F \times \bullet T, \$], [T \rightarrow F \times \bullet T, -],$

$[T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -],$

$[T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -],$

$[F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -],$

$[F \rightarrow \bullet \text{id}, \times] \}$

30

$$I_6 = \{\text{goto}(I_3, \times)\}$$

$$\begin{aligned} & \{[T \rightarrow F \times \bullet T, \$], [T \rightarrow F \times \bullet T, -], \\ & [T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -], \\ & [T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -], \\ & [F \rightarrow \bullet \text{id}, \$], [F \rightarrow \bullet \text{id}, -], \\ & [F \rightarrow \bullet \text{id}, \times] \} \end{aligned}$$

31

$$I_7 = \{\text{goto}(I_5, E)\}$$

$$\{[E \rightarrow T - E \bullet, \$] \}$$

$$I_2 = \{\text{goto}(I_5, T)\}$$

$$I_3 = \{\text{goto}(I_5, F)\}$$

$$I_4 = \{\text{goto}(I_5, \text{id})\}$$

32

$$I_7 = \{\text{goto}(I_5, E)\}$$

$$\{[E \rightarrow T - E \bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_5, T)\}$$

$$I_3 = \{\text{goto}(I_5, F)\}$$

$$I_4 = \{\text{goto}(I_5, \text{id})\}$$

33

$$I_7 = \{\text{goto}(I_5, E)\}$$

$$\{[E \rightarrow T - E \bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_5, T)\}$$

$$I_3 = \{\text{goto}(I_5, F)\}$$

$$I_4 = \{\text{goto}(I_5, \text{id})\}$$

34

$$I_7 = \{\text{goto}(I_5, E)\} \\ \{[E \rightarrow T - E \bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_5, T)\}$$

$$I_3 = \{\text{goto}(I_5, F)\}$$

$$I_4 = \{\text{goto}(I_5, id)\}$$

35

$$I_7 = \{\text{goto}(I_5, E)\} \\ \{[E \rightarrow T - E \bullet, \$]\}$$

$$I_2 = \{\text{goto}(I_5, T)\}$$

$$I_3 = \{\text{goto}(I_5, F)\}$$

$$I_4 = \{\text{goto}(I_5, id)\}$$

36

$$I_8 = \{\text{goto}(I_6, T)\} \\ \{ [T \rightarrow F \times T \bullet, \$], \\ [T \rightarrow F \times T \bullet, -] \}$$

$$I_3 = \{\text{goto}(I_6, F)\}$$

$$I_4 = \{\text{goto}(I_6, \text{id})\}$$

37

$$I_8 = \{\text{goto}(I_6, T)\} \\ \{ [T \rightarrow F \times T \bullet, \$], \\ [T \rightarrow F \times T \bullet, -] \}$$

$$I_3 = \{\text{goto}(I_6, F)\}$$

$$I_4 = \{\text{goto}(I_6, \text{id})\}$$

38

$$I_8 = \{\text{goto}(I_6, T)\}$$

$$\{ [T \rightarrow F \times T \bullet, \$], \\ [T \rightarrow F \times T \bullet, -] \}$$

$$I_3 = \{\text{goto}(I_6, F)\}$$

$$I_4 = \{\text{goto}(I_6, id)\}$$

39

Filling the LR(1) Table

	Action				Goto		
	id	−	×	\$	E	T	F
0							
1							
2							
3							
4							
5							
6							
7							
8							

40

Apply

1. if $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$ then set $\text{Action}[i, a] \leftarrow \text{"shift } j\text{"}$. Here, a is a terminal.

41

$$I_0 = \{ [S' \rightarrow \bullet E, \$], [E \rightarrow \bullet T - E, \$], \\ [E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$], \\ [T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$], \\ [T \rightarrow \bullet F, -], [F \rightarrow \bullet \text{id}, \$], \\ [F \rightarrow \bullet \text{id}, -], [F \rightarrow \bullet \text{id}, \times] \}$$

$$\text{goto}(I_0, \text{id}) = I_4$$

$$\Rightarrow \text{Action}[0, \text{id}] \leftarrow \text{shift } 4$$

42

$I_0 = \{ [S' \rightarrow \bullet E, \$], [E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -], [F \rightarrow \bullet id, \$], [F \rightarrow \bullet id, -], [F \rightarrow \bullet id, \times] \}$

goto(I_0 , id) = I_4

$\Rightarrow \text{Action}[0, id] \leftarrow \text{shift } 4$

43

$I_0 = \{ [S' \rightarrow \bullet E, \$], [E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -], [F \rightarrow \bullet id, \$], [F \rightarrow \bullet id, -], [F \rightarrow \bullet id, \times] \}$

goto(I_0 , id) = I_4

$\Rightarrow \text{Action}[0, id] \leftarrow \text{shift } 4$

44

$$I_0 = \{ [S' \rightarrow \bullet E, \$], [E \rightarrow \bullet T - E, \$], [E \rightarrow \bullet T, \$], [T \rightarrow \bullet F \times T, \$], [T \rightarrow \bullet F \times T, -], [T \rightarrow \bullet F, \$], [T \rightarrow \bullet F, -], [F \rightarrow \bullet id, \$], [F \rightarrow \bullet id, -], [F \rightarrow \bullet id, \times] \}$$

$\text{goto}(I_0, id) = I_4$

$\Rightarrow \text{Action}[0, id] \leftarrow \text{shift } 4$

45

	Action				Goto		
	id	-	×	\$	E	T	F
0							
1							
2							
3							
4							
5							
6							
7							
8							

46

$$I_2 = \{[E \rightarrow T \bullet - E, \$], [E \rightarrow T \bullet, \$]\}$$

$$\text{goto}(I_2, -) = I_5$$

$$\Rightarrow \text{Action}[2, -] \leftarrow \text{shift } 5$$

47

$$I_2 = \{[E \rightarrow T \bullet - E, \$], [E \rightarrow T \bullet, \$]\}$$

$$\text{goto}(I_2, -) = I_5$$

$$\Rightarrow \text{Action}[2, -] \leftarrow \text{shift } 5$$

48

$$I_2 = \{[E \rightarrow T \bullet - E, \$], [E \rightarrow T \bullet, \$]\}$$

$$\text{goto}(I_2, -) = I_5$$

$\Rightarrow \text{Action}[2, -] \leftarrow \text{shift } 5$

49

	Action				Goto		
	id	-	×	\$	E	T	F
0	s4						
1							
2							
3							
4							
5							
6							
7							
8							

50

$I_3 = \{$
 $[T \rightarrow F \bullet \times T, \$], [T \rightarrow F \bullet, \$],$
 $[T \rightarrow F \bullet \times T, -], [T \rightarrow F \bullet, -]\}$



$\text{goto}(I_3, \times) = I_6$

$\Rightarrow \text{Action}[3, \times] \leftarrow \text{shift } 6$

51

$I_3 = \{$
 $[T \rightarrow F \bullet \times T, \$], [T \rightarrow F \bullet, \$],$
 $[T \rightarrow F \bullet \times T, -], [T \rightarrow F \bullet, -]\}$

$\text{goto}(I_3, \times) = I_6$

$\Rightarrow \text{Action}[3, \times] \leftarrow \text{shift } 6$

52

$$I_3 = \{$$

$$[T \rightarrow F \bullet \times T, \$], [T \rightarrow F \bullet, \$],$$

$$[T \rightarrow F \bullet \times T, -], [T \rightarrow F \bullet, -]\}$$

$$\text{goto}(I_3, \times) = I_6$$

$$\Rightarrow \text{Action}[3, \times] \leftarrow \text{shift } 6$$

53

	Action				Goto		
	id	-	×	\$	E	T	F
0	s4						
1							
2		s5					
3							
4							
5							
6							
7							
8							

54

$\text{goto}(I_5, \text{id}) = I_4$

$\Rightarrow \text{Action}[5, \text{id}] \leftarrow \text{shift } 4$

$\text{goto}(I_6, \text{id}) = I_4$

$\Rightarrow \text{Action}[6, \text{id}] \leftarrow \text{shift } 4$

55

$\text{goto}(I_5, \text{id}) = I_4$

$\Rightarrow \text{Action}[5, \text{id}] \leftarrow \text{shift } 4$

$\text{goto}(I_6, \text{id}) = I_4$

$\Rightarrow \text{Action}[6, \text{id}] \leftarrow \text{shift } 4$

56

	Action				Goto		
	id	−	×	\$	E	T	F
0	s4						
1							
2		s5					
3			s6				
4							
5							
6							
7							
8							

57

Apply

- if $[A \rightarrow \alpha\bullet, a] \in I_i$ and $A \neq S'$ then set $\text{action}[i, a]$ to “reduce $A \rightarrow \alpha$ ”

58

$$I_2 = \{ [E \rightarrow T \bullet - E, \$], \\ [E \rightarrow T \bullet, \$] \}$$


$\Rightarrow \text{Action}[2, \$] \leftarrow \text{reduce } 3$

59

$$I_2 = \{ [E \rightarrow T \bullet - E, \$], \\ [E \rightarrow T \bullet, \$] \}$$


$\Rightarrow \text{Action}[2, \$] \leftarrow \text{reduce } 3$

60

	Action				Goto		
	id	−	×	\$	E	T	F
0	s4						
1							
2		s5					
3			s6				
4							
5	s4						
6	s4						
7							
8							

61

$$I_3 = \{$$

$$[T \rightarrow F \bullet \times T, \$], [T \rightarrow F \bullet, \$],$$

$$[T \rightarrow F \bullet \times T, -], [T \rightarrow F \bullet, -] \}$$

$\Rightarrow \text{Action}[3, \$] \leftarrow \text{reduce } 5$

$\Rightarrow \text{Action}[3, -] \leftarrow \text{reduce } 5$

62

$$I_3 = \{ [T \rightarrow F \bullet \times T, \$], [T \rightarrow F \bullet, \$], [T \rightarrow F \bullet \times T, -], [T \rightarrow F \bullet, -] \}$$

$\Rightarrow \text{Action}[3, \$] \leftarrow \text{reduce } 5$

$\Rightarrow \text{Action}[3, -] \leftarrow \text{reduce } 5$

63

		Action			Goto		
	id	-	$\times$	\$	E	T	F
0	s4						
1							
2		s5		r3			
3			s6				
4							
5	s4						
6	s4						
7							
8							

64

$$I_4 = \{ [F \rightarrow id\bullet, \$], [F \rightarrow id\bullet, -], [F \rightarrow id\bullet, \times] \}$$

$\Rightarrow \text{Action}[4, \$] \leftarrow \text{reduce } 6$

$\Rightarrow \text{Action}[4, -] \leftarrow \text{reduce } 6$

$\Rightarrow \text{Action}[4, \times] \leftarrow \text{reduce } 6$

65

$$I_4 = \{ [F \rightarrow id\bullet, \$], [F \rightarrow id\bullet, -], [F \rightarrow id\bullet, \times] \}$$

$\Rightarrow \text{Action}[4, \$] \leftarrow \text{reduce } 6$

$\Rightarrow \text{Action}[4, -] \leftarrow \text{reduce } 6$

$\Rightarrow \text{Action}[4, \times] \leftarrow \text{reduce } 6$

66

	Action				Goto		
	id	−	×	\$	E	T	F
0	s4						
1							
2		s5		r3			
3		r5	s6	r5			
4							
5	s4						
6	s4						
7							
8							

67

$$I_7 = \{ [E \rightarrow T - E\bullet, \$] \}$$

$$\Rightarrow \text{Action}[7, \$] \leftarrow \text{reduce } 2$$

$$I_8 = \{ [T \rightarrow F \times T\bullet, \$], \\ [T \rightarrow F \times T\bullet, -] \}$$

$$\Rightarrow \text{Action}[8, \$] \leftarrow \text{reduce } 4$$

$$\Rightarrow \text{Action}[8, -] \leftarrow \text{reduce } 4$$

68

$$I_7 = \{[E \rightarrow T - E\bullet, \$]\}$$

$$\Rightarrow \text{Action}[7, \$] \leftarrow \text{reduce } 2$$

$$I_8 = \{ \begin{array}{l} [T \rightarrow F \times T\bullet, \$], \\ [T \rightarrow F \times T\bullet, -] \end{array} \}$$

$$\Rightarrow \text{Action}[8, \$] \leftarrow \text{reduce } 4$$

$$\Rightarrow \text{Action}[8, -] \leftarrow \text{reduce } 4$$

69

$$I_7 = \{[E \rightarrow T - E\bullet, \$]\}$$

$$\Rightarrow \text{Action}[7, \$] \leftarrow \text{reduce } 2$$

$$I_8 = \{ \begin{array}{l} [T \rightarrow F \times T\bullet, \$], \\ [T \rightarrow F \times T\bullet, -] \end{array} \}$$

$$\Rightarrow \text{Action}[8, \$] \leftarrow \text{reduce } 4$$

$$\Rightarrow \text{Action}[8, -] \leftarrow \text{reduce } 4$$

70

	Action				Goto		
	id	−	×	\$	E	T	F
0	s4						
1							
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4						
6	s4						
7							
8							

71

Apply

3. if $[S' \rightarrow S\bullet, \$] \in I_i$ then
 set $\text{action}[i, \$]$ to “accept”

72

$$I_1 = \{[S' \rightarrow E\bullet, \$] \}$$

$\Rightarrow \text{Action}[1, \$] \leftarrow \text{accept}$

73

	Action				Goto		
	id	−	×	\$	E	T	F
0	s4						
1							
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4						
6	s4						
7				r2			
8		r4		r4			

74

Apply

for each non-terminal $A \in G$

- if $\text{goto}(I_i, A) = I_j$ then
 $\text{goto}[i, A] \leftarrow j$.

75

$$\text{goto}(I_0, E) = I_1 \Rightarrow \text{goto}[0, E] \leftarrow 1$$

$$\text{goto}(I_0, T) = I_2 \Rightarrow \text{goto}[0, T] \leftarrow 2$$

$$\text{goto}(I_0, F) = I_3 \Rightarrow \text{goto}[0, F] \leftarrow 3$$

$$\text{goto}(I_5, E) = I_7 \Rightarrow \text{goto}[5, E] \leftarrow 7$$

$$\text{goto}(I_5, T) = I_2 \Rightarrow \text{goto}[5, T] \leftarrow 2$$

$$\text{goto}(I_5, F) = I_3 \Rightarrow \text{goto}[5, F] \leftarrow 3$$

$$\text{goto}(I_6, T) = I_8 \Rightarrow \text{goto}[6, T] \leftarrow 8$$

$$\text{goto}(I_6, F) = I_3 \Rightarrow \text{goto}[6, F] \leftarrow 3$$

76

	Action				Goto		
	id	–	×	\$	E	T	F
0	s4						
1				acc			
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4						
6	s4						
7				r2			
8		r4		r4			

77

Example Parse

- Let us parse the expression $x - y \times z$ using the LR(1) table
- The scanner will encode the input string as $id - id \times id \$$ where $\$$ is the EOF marker

78

Example Parse

- Let us parse the expression $x - y \times z$ using the LR(1) table
- The scanner will encode the input string as **id – id × id \$** where **\$** is the EOF marker

79

Compiler Construction

Sohail Aslam

Lecture 28

Stack	Input	
⌈0	id – id × id \$	

2

	Action				Goto		
	id	–	×	\$	E	T	F
0	s4				1	2	3
1				acc			
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4				7	2	3
6	s4					8	3
7				r2			
8		r4		r4			

3

Stack	Input	
⌈0	id – id × id \$	

4

	Action				Goto		
	id	–	×	\$	E	T	F
0	s4				1	2	3
1				acc			
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4				7	2	3
6	s4					8	3
7				r2			
8		r4		r4			

5

Stack	Input	
⌈0	id – id × id \$	s4
⌈0id4	– id × id \$	r6 F → id
⌈0F3		

6

Stack	Input	
⌈0	id – id × id \$	s4
⌈0id4	– id × id \$	r6 F → id
⌈0F3		

7

Stack	Input	
⌈0T2–5id4	× id \$	

Stack	Input	
⌈0T2–5T2	\$	

LR(1) Skeleton Parser

```
stack.push(dummy); stack.push(0);
done = false; token = scanner.next();
while (!done) {
    s = stack.top();
    if( Action[s,token] == "reduce  $A \rightarrow \beta$ ") {
        stack.pop(2×| $\beta$ |);
        s = stack.top();
        stack.push(A);
        stack.push(Goto[s,A]);
    }
```

10

```
    else if( Action[s,token] == "shift i"){
        stack.push(token); stack.push(i);
        token = scanner.next();
    }
    else if(Action[s,token] == "accept"
            && token == "$" )
        done = true;
    else
        report error and recover;
}
report success;
```

11

Shift/Reduce Conflicts

- If a DFA states contains both $[X \rightarrow \alpha \bullet a \beta, b]$ and $[Y \rightarrow \gamma \bullet, a]$
- Then on input “a” we could either
 - shift into state $[X \rightarrow \alpha a \bullet \beta, b]$, or
 - reduce with $Y \rightarrow \gamma$

12

Shift/Reduce Conflicts

- If a DFA states contains both $[X \rightarrow \alpha \bullet a \beta, b]$ and $[Y \rightarrow \gamma \bullet, a]$
- Then on input “a” we could either
 - shift into state $[X \rightarrow \alpha a \bullet \beta, b]$, or
 - reduce with $Y \rightarrow \gamma$

13

Shift/Reduce Conflicts

- If a DFA states contains both $[X \rightarrow \alpha \bullet a \beta, b]$ and $[Y \rightarrow \gamma \bullet, a]$
- Then on input “**a**” we could either
 - **shift** into state $[X \rightarrow \alpha a \bullet \beta, b]$, or
 - **reduce** with $Y \rightarrow \gamma$

14

Shift/Reduce Conflicts

- If a DFA states contains both $[X \rightarrow \alpha \bullet a \beta, b]$ and $[Y \rightarrow \gamma \bullet, a]$
- Then on input “**a**” we could either
 - **shift** into state $[X \rightarrow \alpha a \bullet \beta, b]$, or
 - **reduce** with $Y \rightarrow \gamma$

15

Shift/Reduce Conflicts

- This is called a *shift-reduce conflict*
- Typically, this is due to ambiguities in the grammar

16

Shift/Reduce Conflicts

- This is called a *shift-reduce conflict*
- Typically, this is due to **ambiguities** in the grammar

17

Shift/Reduce Conflicts

Classic example: dangling **else**

stmt $\rightarrow$ if **E** then **stmt**
 | if **E** then **stmt** else **stmt**

18

Shift/Reduce Conflicts

We will have DFA state
containing

[**stmt** $\rightarrow$ if **E** then **stmt**•, else]

[**stmt** $\rightarrow$ if **E** then **stmt** •else **stmt**, x]

19

Shift/Reduce Conflicts

If **else** follows, we can shift

$[stmt \rightarrow \underline{if} \ E \ \underline{then} \ stmt \bullet \ \underline{else} \ stmt, x]$

or reduce

$[stmt \rightarrow \underline{if} \ E \ \underline{then} \ stmt \bullet, \ \underline{else}]$

20

Shift/Reduce Conflicts

If **else** follows, we can shift

$[stmt \rightarrow \underline{if} \ E \ \underline{then} \ stmt \bullet \ \underline{else} \ stmt, x]$

or reduce

$[stmt \rightarrow \underline{if} \ E \ \underline{then} \ stmt \bullet, \ \underline{else}]$

21

Shift/Reduce Conflicts

[stmt $\rightarrow$ if E then stmt•, else]

[stmt $\rightarrow$ if E then stmt• else stmt, x]

Typical action is **shift** so that **else** matches with most recent **if**

22

Compiler Construction

Sohail Aslam

Lecture 29

Shift/Reduce Conflicts

- Consider the ambiguous grammar

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

- We will DFA state containing
[$E \rightarrow E \times E \bullet$, +]
[$E \rightarrow E \bullet + E$, +]

2

Shift/Reduce Conflicts

- Consider the ambiguous grammar

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

- We will DFA state containing
[$E \rightarrow E \times E \bullet$, +]
[$E \rightarrow E \bullet + E$, +]

3

$[E \rightarrow E \times E \bullet, +]$

$[E \rightarrow E \bullet + E, +]$

- Again we have a shift/reduce conflict
- We need to reduce because $\times$ has precedence over $+$

4

$[E \rightarrow E \times E \bullet, +]$

$[E \rightarrow E \bullet + E, +]$

- Again we have a shift/reduce conflict
- We need to **reduce** because $\times$ has *precedence* over $+$

5

Reduce/Reduce Conflicts

- If a DFA states contains both
 $[X \rightarrow \alpha \bullet, a]$ and $[Y \rightarrow \beta \bullet, a]$

Then on input “**a**” we don’t know which production to reduce

6

Reduce/Reduce Conflicts

- This is called a *reduce-reduce conflict*
- Usually due to **gross ambiguity** in the grammar

7

LR(1) Table Size

- LR(1) parsing table for even a simple language can be **extremely large** with **thousands** of entries.

8

LR(1) Table Size

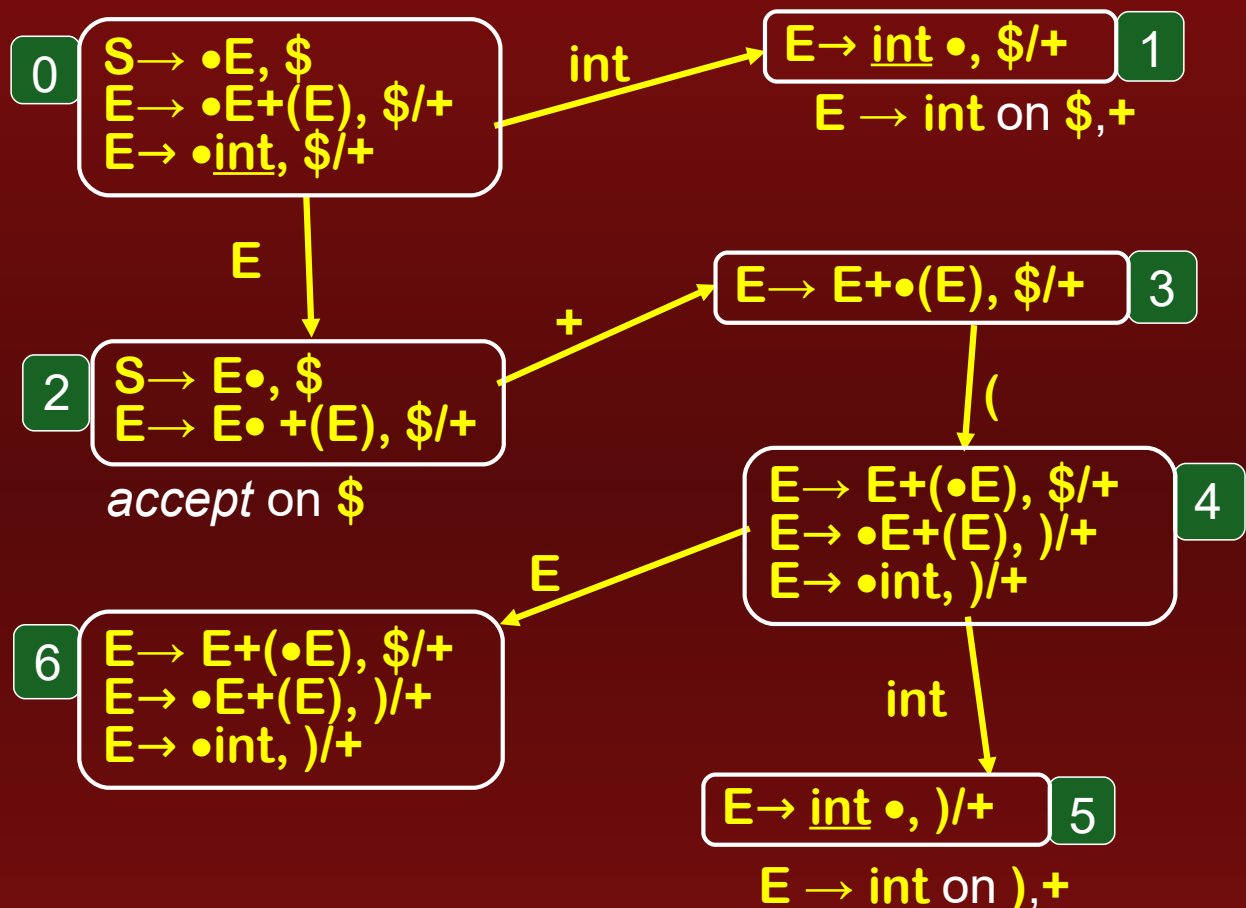
- It is possible to **reduce** the **size** of the **table**
- Many states in the DFA are similar

9

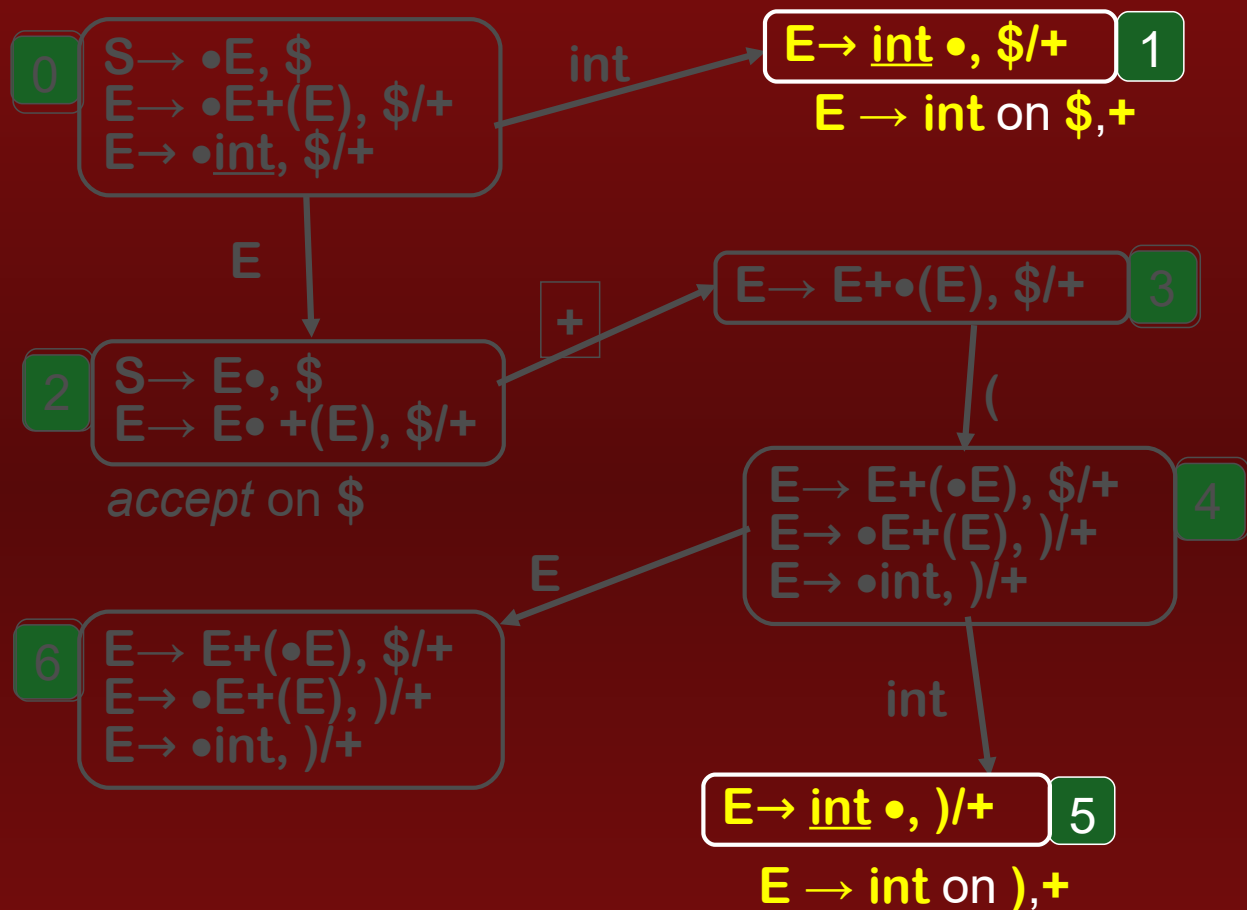
LR(1) Table Size

- It is possible to **reduce** the **size** of the **table**
- Many states in the DFA are **similar**

10



11



12

LR(1) Table Size

These states have the same *core*

E → int •, \$/+ **1**
E → int on \$,+

E → int •,)/+ **5**
E → int on),+

13

LR(1) Table Size

Upon **merge**, we obtain

$$\boxed{E \rightarrow \underline{\text{int}} \bullet, \$ / + /)} \quad 1'$$

$E \rightarrow \text{int}$ on $\$, +,)$

14

The Core

Definition:

The core of set of LR items is the set of **first** components

- *without the lookahead terminals*

15

The Core

Example:

the core of

$$\{ [X \rightarrow \alpha \bullet \beta, b], [Y \rightarrow \gamma \bullet \delta, d] \}$$

is

$$\{ X \rightarrow \alpha \bullet \beta, Y \rightarrow \gamma \bullet \delta \}$$

16

Core of a Set of LR Items

- Consider the LR(1) states

$$\{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c] \}$$

$$\{ [X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d] \}$$

- They have the same core and can be merged

17

Core of a Set of LR Items

- Consider the LR(1) states
$$\{ [X \rightarrow \alpha\bullet, a], [Y \rightarrow \beta\bullet, c] \}$$
$$\{ [X \rightarrow \alpha\bullet, b], [Y \rightarrow \beta\bullet, d] \}$$
- They have the **same core** and can be merged

18

LALR States

- The merged state contains
$$\{ [X \rightarrow \alpha\bullet, a/b], [Y \rightarrow \beta\bullet, c/d] \}$$
- These are called the *LALR(1) states*

19

LALR States

- The merged state contains
$$\{ [X \rightarrow \alpha\bullet, a/b], [Y \rightarrow \beta\bullet, c/d] \}$$
- These are called the *LALR(1) states*

20

LALR States

- LALR(1) stands for LookAhead LR(1)
- leads to tables that have 10 times **fewer** states than LR(1)

21

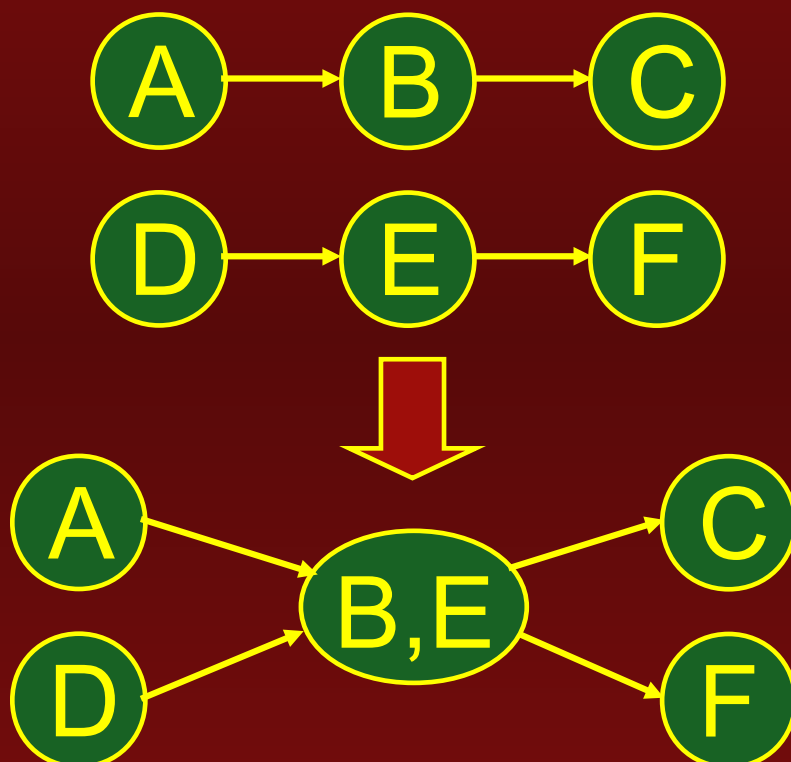
A LALR(1) DFA

Repeat until all states have distinct code

- choose two distinct states with same core
- merge states by creating a new one with the union of all the items
- point edges from predecessors to new state
- new state points to all the previous successors

22

A LALR(1) DFA



23

LALR(1) parsing

- LALR languages are not natural
 - *they are an efficiency hack on LR languages*
- Any reasonable programming language has a LALR(1) grammar

24

LALR(1) parsing

- LALR languages are not natural
 - *they are an efficiency hack on LR languages*
- Any reasonable programming language has a LALR(1) grammar

25

LALR(1) parsing

- LALR(1) has become a **standard** for programming languages and for **parser generators**
- Parser generators exist for LL(1) and LALR(1) grammars

26

Parser Generators

- LALR(1) - **YACC, Bison, CUP**
- LL(1) – **ANTLR**
- Recursive Descent - **JavaCC**

27

Compiler Construction

Sohail Aslam

Lecture 30

Parser Generators

- **YACC** – Yet Another Compiler Compiler appeared in 1975 as a **Unix** application.
- The other **companion** application **Lex** appeared at the same time.

Parser Generators

- These two greatly **aided** the construction of compilers and interpreters.

3

YACC Parser Generator

- The **input** to YACC consists of a specification text **file**.
- The structure of the file is
 - definitions*
 - %%*
 - rules*
 - %%*
 - C/C++ functions*

4

YACC Parser Generator

- The **input** to YACC consists of a specification text **file**.
- The structure of the file is
 - definitions*
 - %%**
 - rules*
 - %%**
 - C/C++ functions*

5

YACC file for a calculator

```
→ %token NUMBER LPAREN RPAREN
%token PLUS MINUS TIMES DIVIDE
%%
expr : expr PLUS expr
    | expr MINUS expr
    | expr TIMES expr
    | expr DIVIDE expr
    | LPAREN expr RPAREN
    | MINUS expr
    | NUMBER
    ;
%%
```

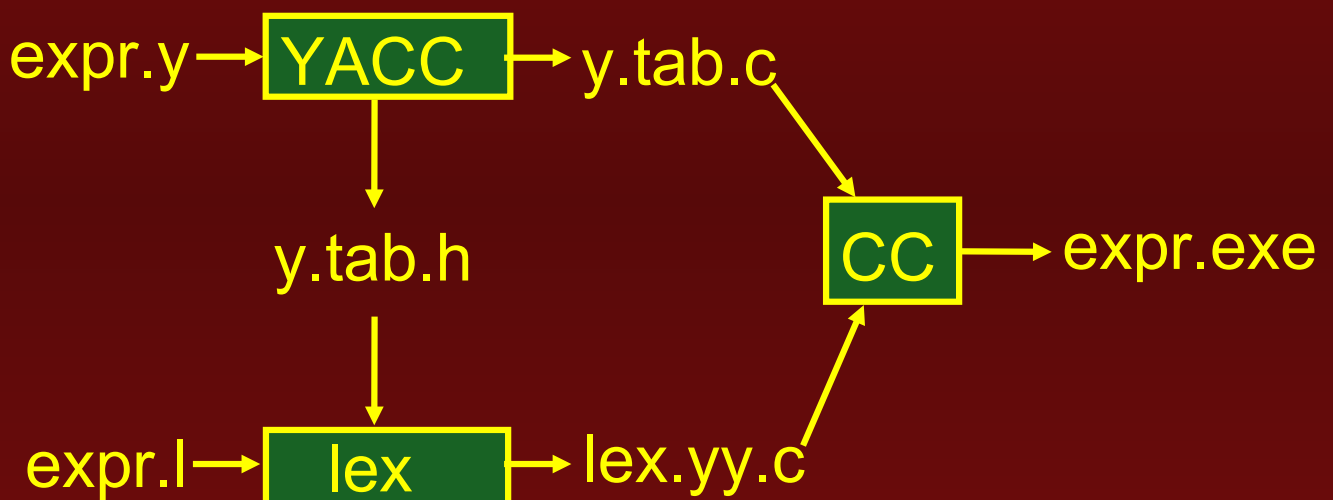
6

Flex input file for a calculator

```
→ %{
#include "y.tab.h"
}%
digit      [0-9]
ws         [ \t\n]+
%%
{ws}       ;
{digit}+   {return NUMBER;}
"+"        {return PLUS;}
"*"        {return TIMES;}
"/"        {return DIVIDE;}
"-"        {return MINUS;}
%%
```

7

Building a parser with YACC and Lex



8

Context Sensitive Analysis

Beyond Syntax

```
void fie(int a, int b)
{ .... }
void fee() {
    int f[3],g[0],h,i,j,k;
    char* p;
    fie(h, i, "ab");
    k = f*i+j;
    h = g[17];
    p = 10;
}
```

what is wrong with this program?

Beyond Syntax

```
void fie(int a, int b)
{ .... }
void fee() {
    int f[3],g[1],h,i,j,k;
    char* p;
    fie(h, i, "ab");
    k = f*i+j;
    h = g[17];
    p = 10;
}
```

dimension of g is 1,
index is 17

11

Beyond Syntax

```
void fie(int a, int b)
{ .... }
void fee() {
    int f[3],g[1],h,i,j,k;
    char* p;
    fie(h, i, "ab");
    k = f*i+j;
    h = g[17];
    p = 10;
}
```

wrong number of
args to function fie

12

Beyond Syntax

```
void fie(int a, int b)
{ .... }
void fee() {
    int f[3],g[1],h,i,j,k;
    char* p;
    fie(h, i, "ab")
    k = f*i+j;
    h = g[17];
    p = 10;
}
```

f is an array;
used without
index

13

Beyond Syntax

```
void fie(int a, int b)
{ .... }
void fee() {
    int f[3],g[1],h,i,j,k;
    char* p;
    fie(h, i, "ab")
    k = f*i+j;
    h = g[17];
    p = 10;
}
```

10 is not a
character string

14

Beyond Syntax

To *generate code*, the compiler needs to answer many questions

15

Beyond Syntax

- Is “**x**” a scaler, an array or a function?
- Is “**x**” declared before it is used?
- Is the expression “**x*y+z**” type-consistent?

16

Beyond Syntax

- Is “**x**” a scaler, an array or a function?
- Is “**x**” declared before it is used?
- Is the expression “**x*y+z**” type-consistent?

17

Beyond Syntax

- Is “**x**” a scaler, an array or a function?
- Is “**x**” declared before it is used?
- Is the expression “**x*y+z**” type-consistent?

18

Beyond Syntax

- In “**a**[**i**,**j**,**k**]” does **a** have three dimensions?
- Does “***p**” reference the result of “**new**”?
- Do “**p**” and “**q**” refer to the same memory location?

19

Beyond Syntax

- In “**a**[**i**,**j**,**k**]” does **a** have three dimensions?
- Does “***p**” reference the result of “**new**”?
- Do “**p**” and “**q**” refer to the same memory location?

20

Beyond Syntax

- In “**a**[**i**,**j**,**k**]” does **a** have three dimensions?
- Does “***p**” reference the result of “**new**”?
- Do “**p**” and “**q**” refer to the same memory location?

21

Compiler Construction

Sohail Aslam

Lecture 31

Beyond Syntax

- These questions are part of **context-sensitive analysis**

not parts of speech

- Answers may involve computation

2

Beyond Syntax

- These questions are part of **context-sensitive analysis**

- Answers depend on **values**, not parts of speech

- Answers may involve computation

3

Beyond Syntax

- These questions are part of **context-sensitive analysis**
- Answers depend on **values**, not parts of speech
- Answers may involve **computation**

4

Beyond Syntax

How can we answer these questions?

- Context-sensitive grammars
- Attribute grammars

5

Beyond Syntax

How can we answer these questions?

- Use formal methods
 - Context-sensitive grammars
 - Attribute grammars

6

Beyond Syntax

How can we answer these questions?

- Use formal methods
 - Context-sensitive grammars
 - Attribute grammars

7

Beyond Syntax

- Use ad-hoc techniques
 - Symbol tables
 - ad-hoc code

8

Attribute Grammars

- A CFG is augmented with a *set of rules*
- Each symbol in the derivation has a set of *attributes*
- Rules specify how to *compute* a value for each attribute

9

Attribute Grammars

- A CFG is **augmented** with a *set of rules*
- Each **symbol** in the derivation has a set of values or *attributes*
- Rules specify how to *compute* a value for each attribute

10

Attribute Grammars

- A CFG is **augmented** with a *set of rules*
- Each **symbol** in the derivation has a set of values or *attributes*
- Rules specify how to *compute* a **value** for each **attribute**

11

Attribute Grammars

- grammar for *signed binary numbers (SBN)*

Number $\rightarrow$ **Sign List**

Sign $\rightarrow$ **+ | -**

List $\rightarrow$ **List Bit | Bit**

Bit $\rightarrow$ **0 | 1**

12

Attribute Grammars

derivation for “-1”

Number $\rightarrow$ **Sign List**

$\rightarrow$ **- List**

$\rightarrow$ **- Bit**

$\rightarrow$ **- 1**

13

Attribute Grammars

For “-101”

Number → Sign List
→ Sign List Bit
→ Sign List 1
→ Sign List Bit 1
→ Sign List 0 1
→ Sign Bit 0 1
→ Sign 1 0 1
→ - 1 0 1

14

Attribute Grammars

For an attributed version of *SBN*, the following attributes are needed

<i>Symbol</i>	<i>Attributes</i>

15

Attribute Grammars

- We will add rules to **compute decimal value** of a signed binary number

16

Attribute Grammars

<i>Productions</i>	<i>Attribution Rules</i>

17

Attribute Grammars

<i>Productions</i>	<i>Attribution Rules</i>
Number $\rightarrow$ Sign List	List.pos $\leftarrow$ 0 if Sign.neg then Number.val $\leftarrow$ - List.val else Number.val $\leftarrow$ List.val

18

Attribute Grammars

<i>Productions</i>	<i>Attribution Rules</i>

19

Attribute Grammars

- Attributes are associated with nodes in parse tree
- Rules are value assignments associated with productions

20

Attribute Grammars

- Attributes are associated with nodes in parse tree
- Rules are value assignments associated with productions

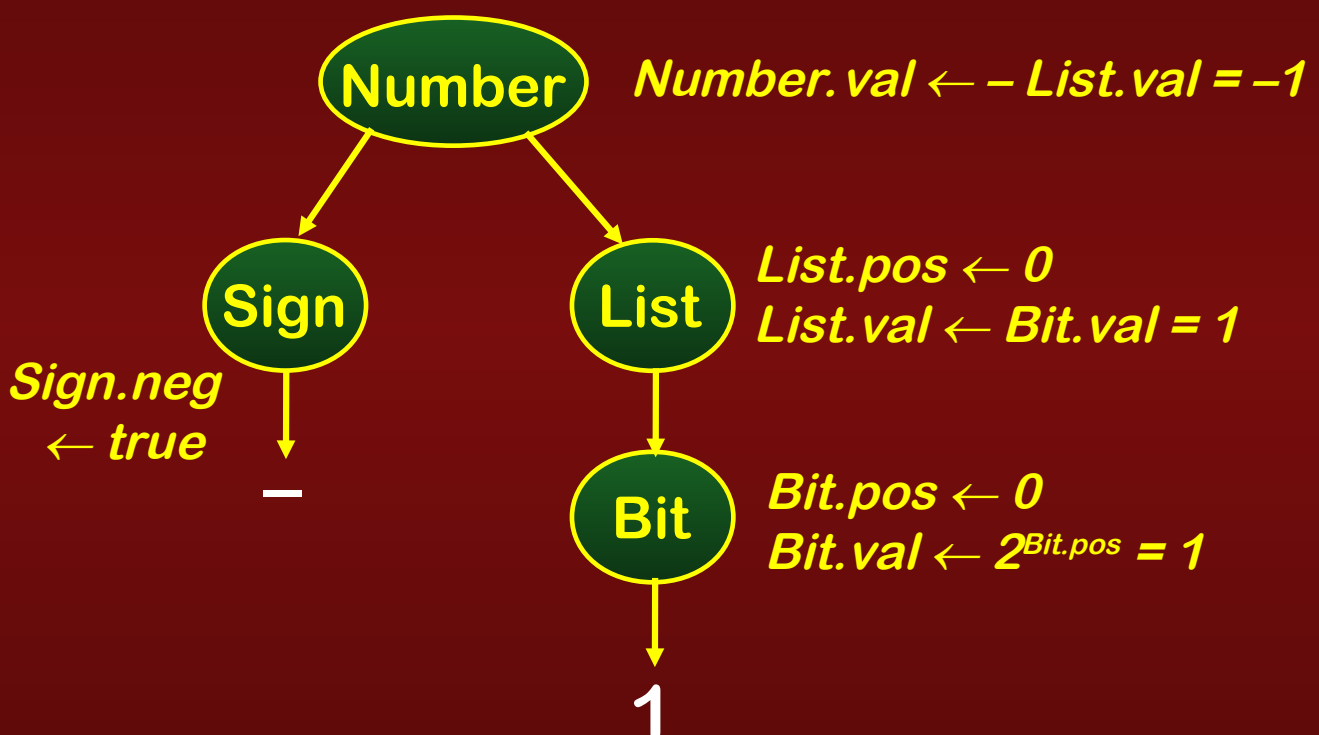
21

Attribute Grammars

- Rules and parse tree define an **attribute dependence graph**
 - Graph must be acyclic

22

Attribute Grammars



23

Attributes

Attributes are distinguished based on the **direction** of **value flow**

1. Synthesized attributes
2. Inherited attributes

24

Synthesized Attributes

- **Attributes** of a node whose values are **defined** wholly in terms of attributes of **node's children** and from **constants** are called ***synthesized attributes***

25

Synthesized Attributes

- Values used to compute synthesized attributes **flow bottom-up** in the parse tree
- Good match to **LR parsing**

26

Inherited Attributes

- Attributes whose values are **defined** in terms of
 - node's own attributes,
 - node's siblings
 - node's parent

laterally in the parse tree

27

Inherited Attributes

- Attributes whose values are **defined** in terms of
 - node's own attributes,
 - node's siblings
 - node's parent
- Values flow **top-down** and **laterally** in the parse tree

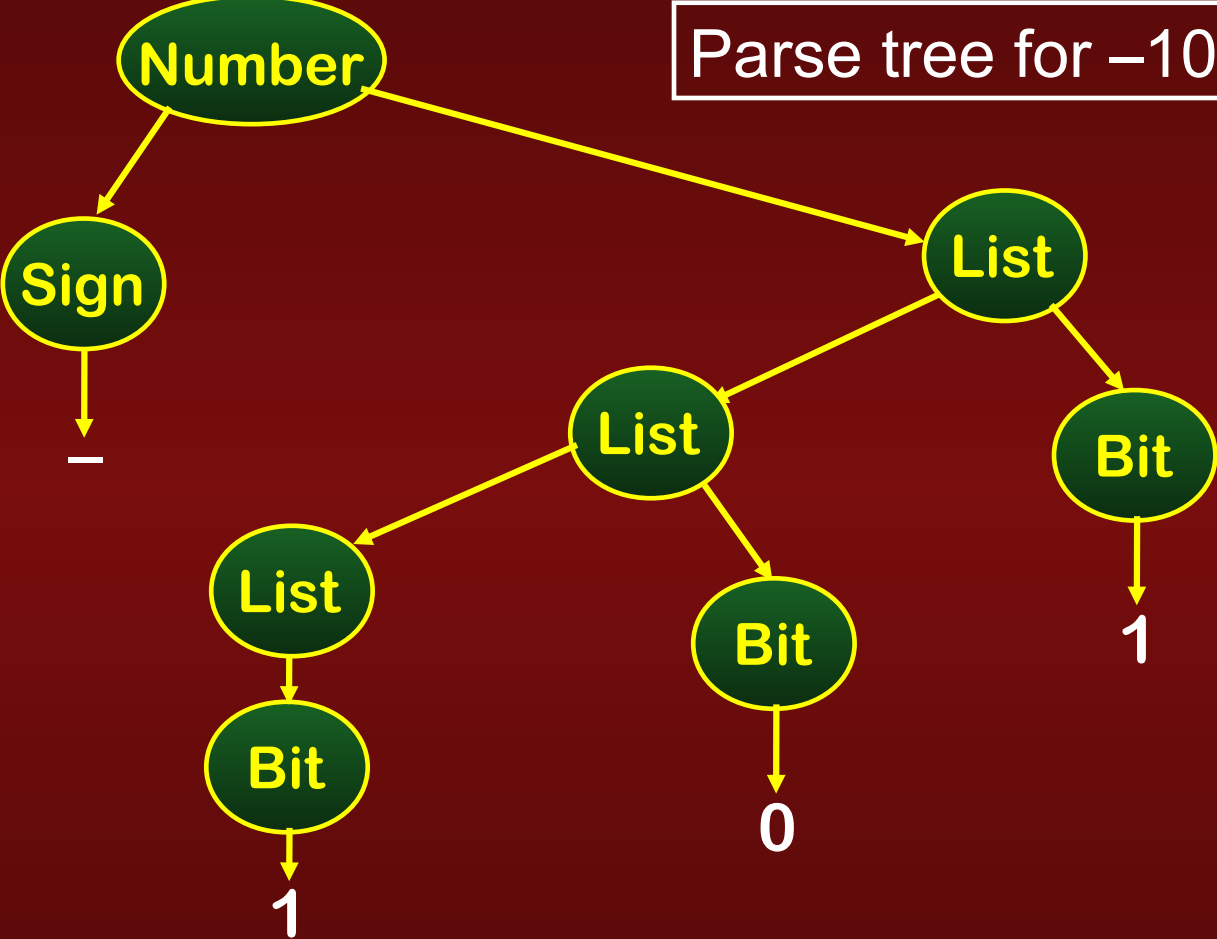
28

Compiler Construction

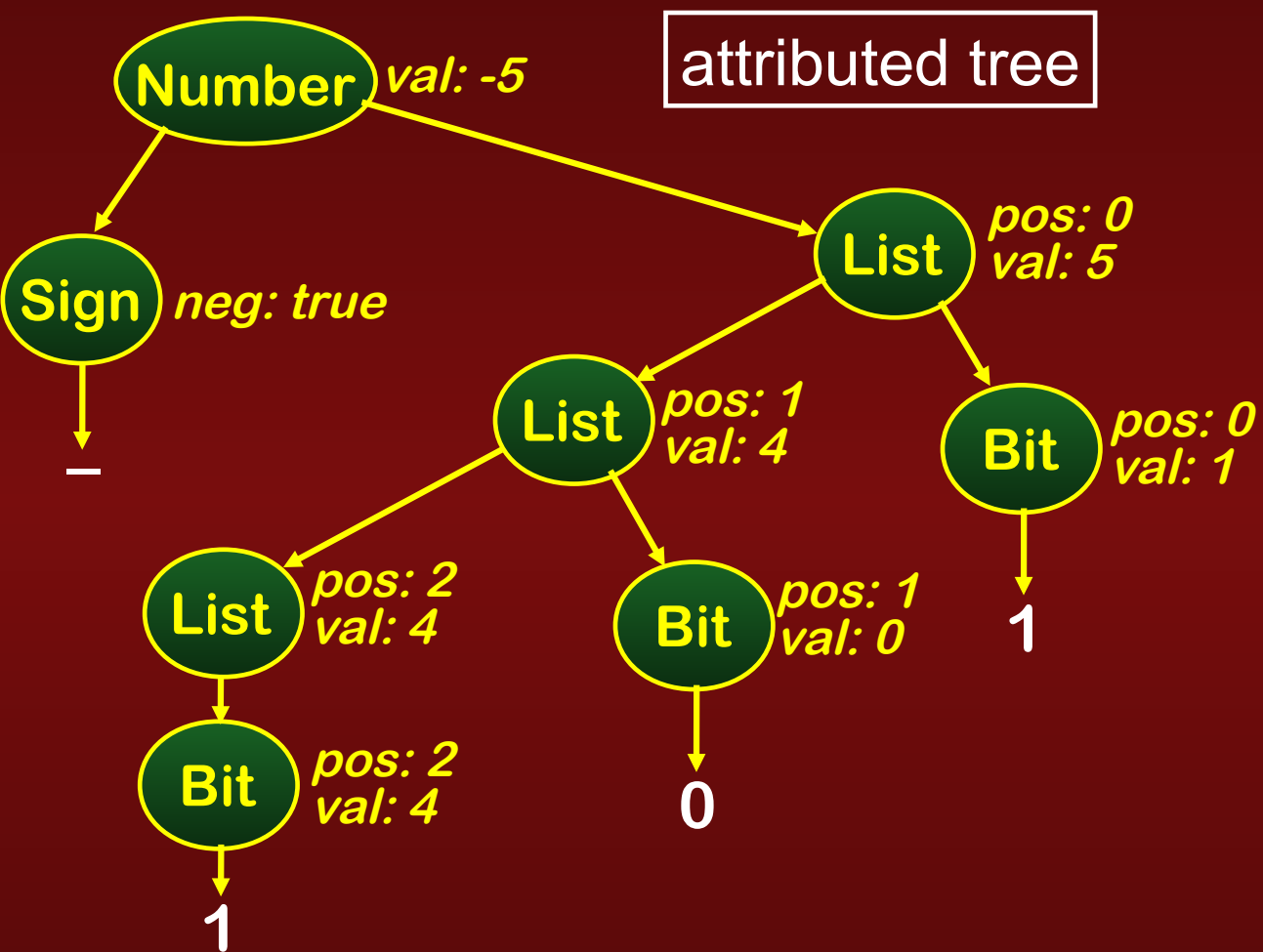
Sohail Aslam

Lecture 32

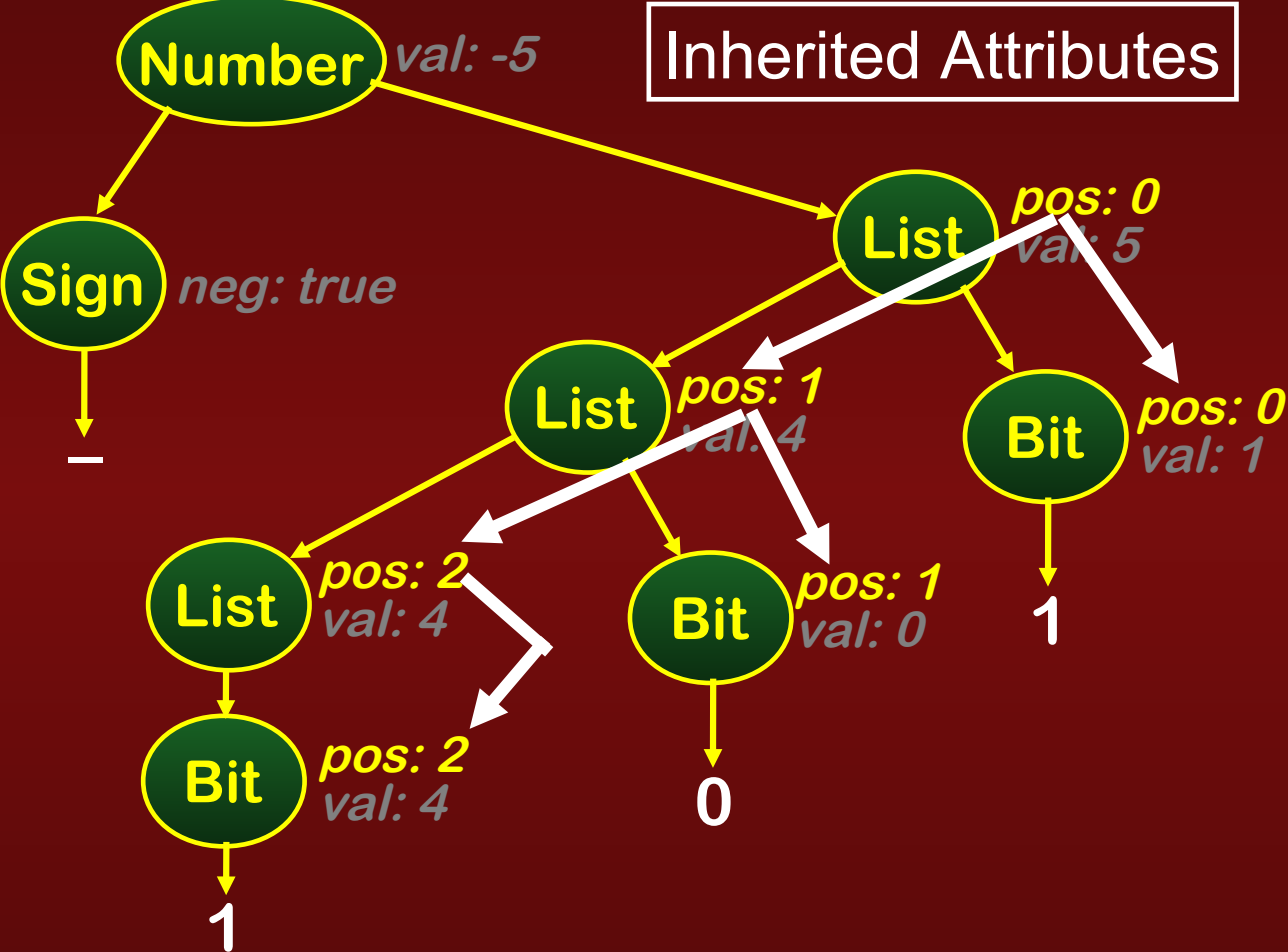
Parse tree for -101



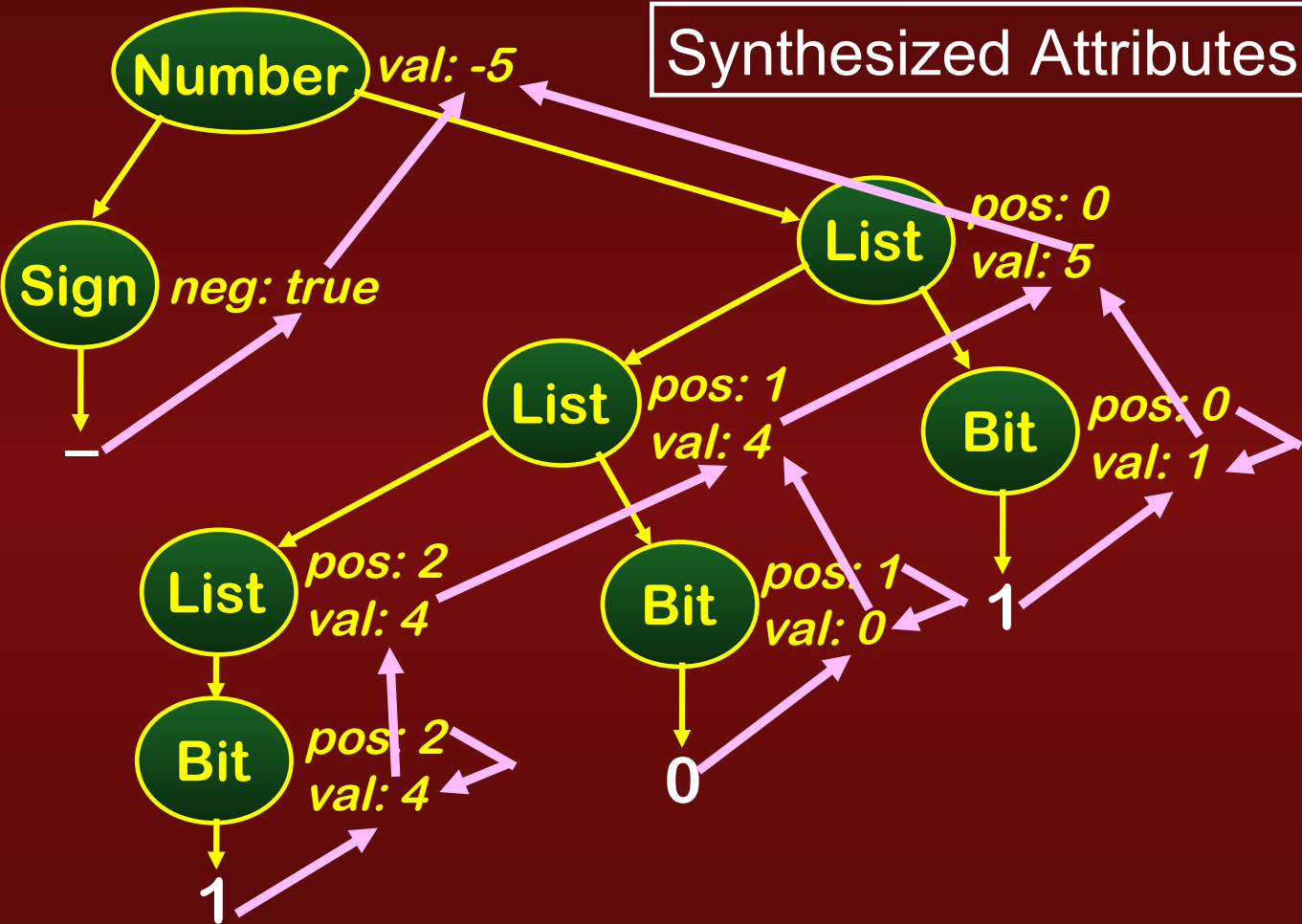
attributed tree

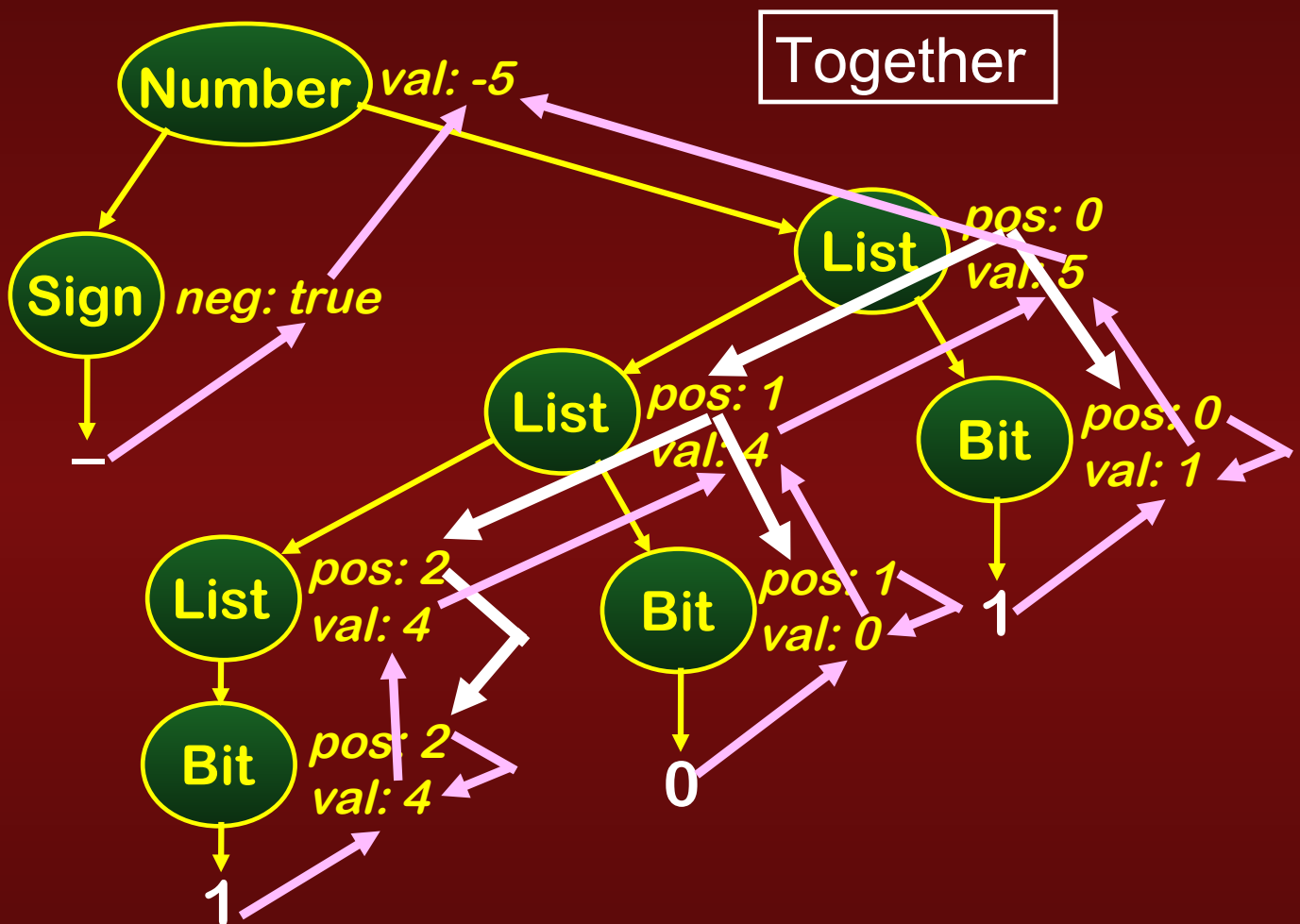


Inherited Attributes

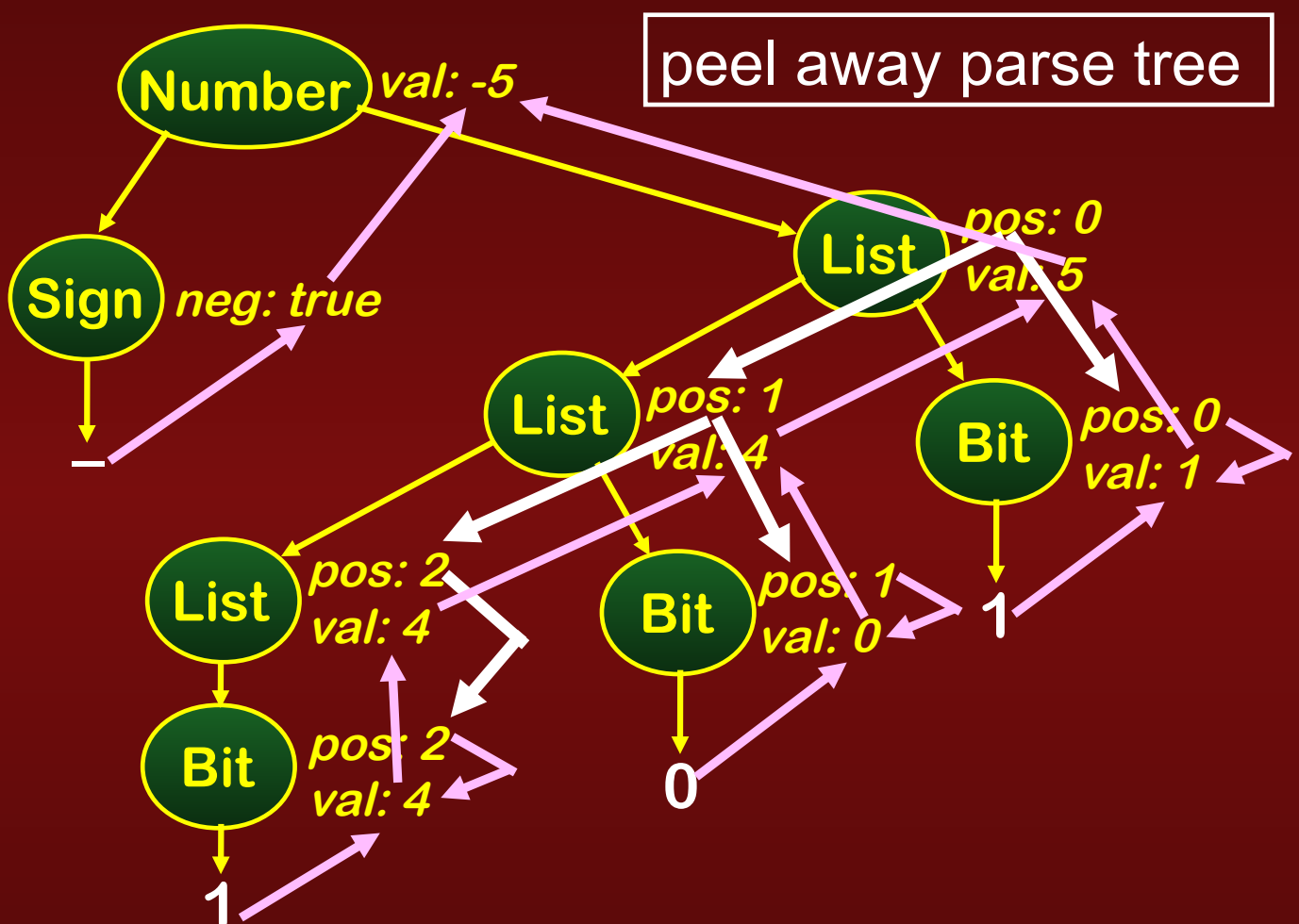


Synthesized Attributes

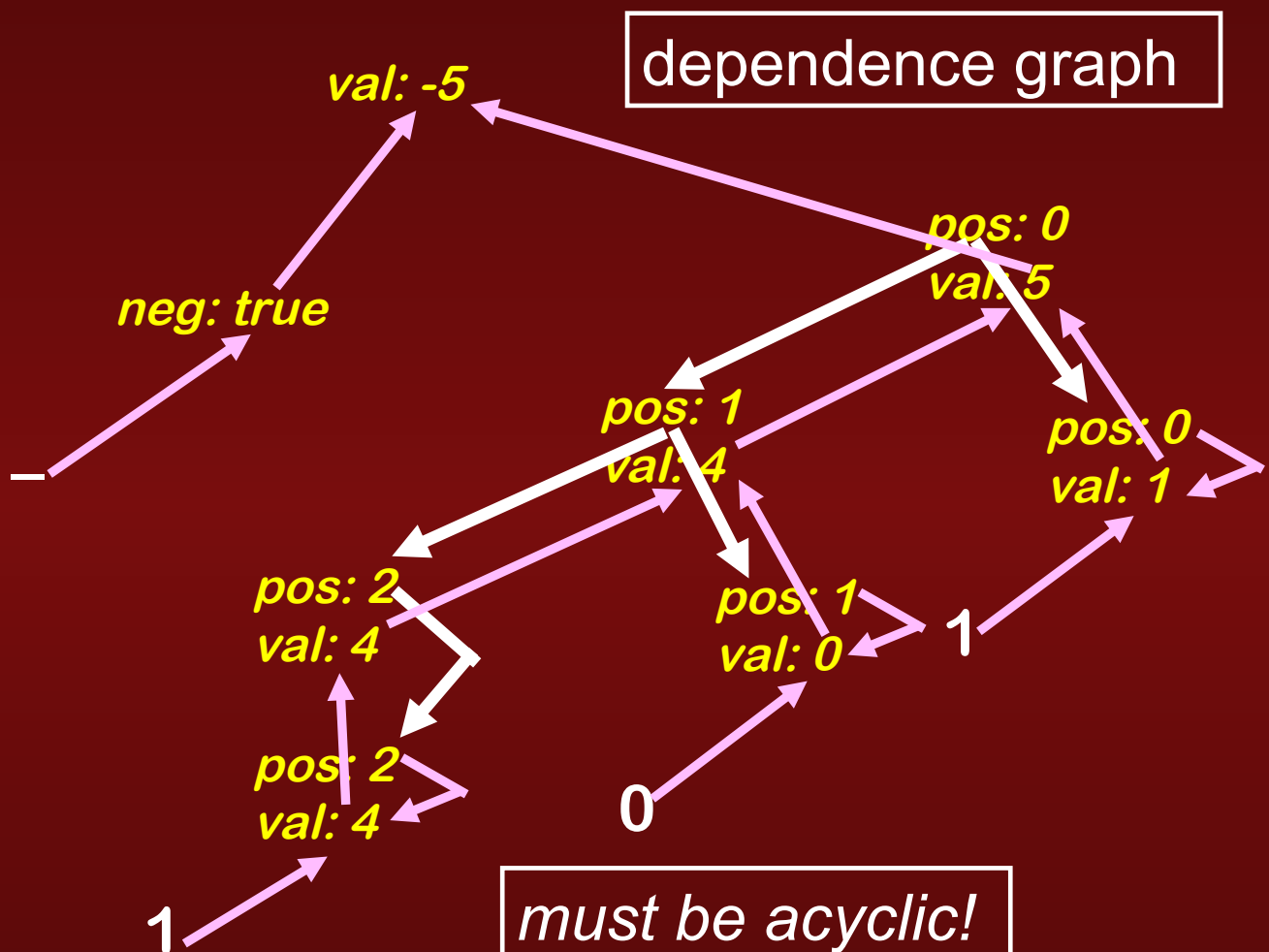
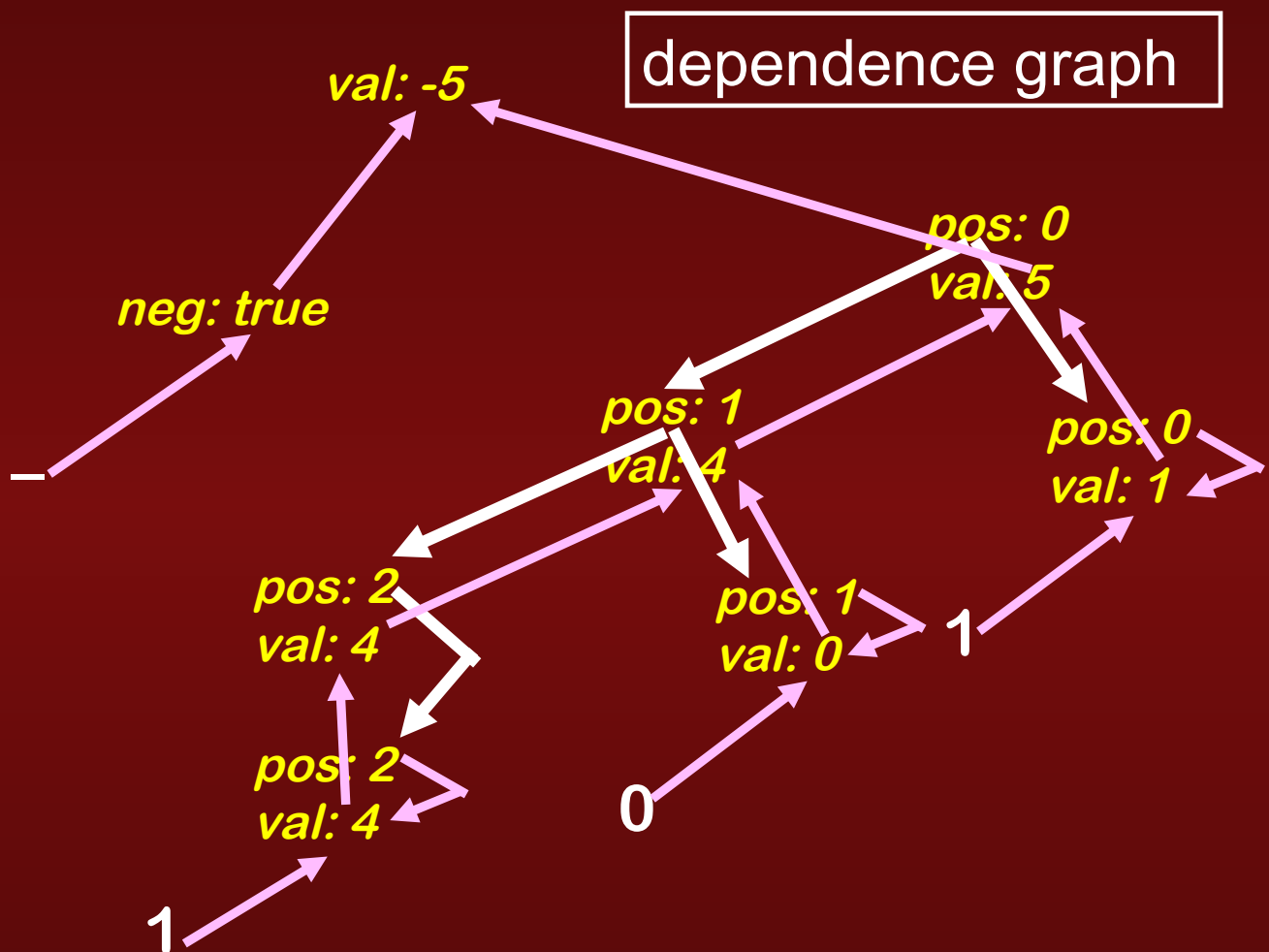




6



7



Evaluation Methods

Dynamic methods

- Build the parse tree
 - Build the dependence graph
 - Topological sort the graph
- order

10

Evaluation Methods

Dynamic methods

- Build the parse tree
 - Build the dependence graph
 - Topological sort the graph
- order

11

Evaluation Methods

Dynamic methods

- Build the parse tree
- Build the dependence graph
- Topological sort the graph
- Order the nodes

12

Evaluation Methods

Dynamic methods

- Build the parse tree
- Build the dependence graph
- Topological sort the graph
- Order the nodes

13

Evaluation Methods

Dynamic methods

- Build the parse tree
- Build the dependence graph
- Topological sort the graph
- Define attributes in topological order

14

Evaluation Methods

Rule-based (treewalk)

- Analyze attribute rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

15

Evaluation Methods

Oblivious (passes, dataflow)

- Ignore rules and parse tree
- Pick a convenient order (at design time) and use it

16

Problems

Attribute grammars have not achieved widespread use due to a myraid of problems

17

Problems

- non-local computation
- traversing parse tree
- storage management for short-lived attributes
- lack of high-quality inexpensive tools

18

Problems

- non-local computation
- traversing parse tree
- storage management for short-lived attributes
- lack of high-quality inexpensive tools

19

Problems

- non-local computation
- traversing parse tree
- storage management for short-lived attributes
- lack of high-quality inexpensive tools

20

Problems

- non-local computation
- traversing parse tree
- storage management for short-lived attributes
- lack of high-quality inexpensive tools

21

Ad-Hoc Analysis

- In **rule-based** evaluators, a sequence of **actions** are associated with grammar **productions**

22

Ad-Hoc Analysis

- Organizing actions required for context-sensitive analysis around structure of the grammar leads to powerful, albeit **ad-hoc**, approach which is used on most parsers

23

Ad-Hoc Analysis

- A **snippet of code** (*action*) is associated with each production that **executes** at parse time
- In top-down parsers, the snippet is added to the appropriate parsing routine

24

Ad-Hoc Analysis

- A **snippet of code** (*action*) is associated with each production that **executes** at parse time
- In **top-down** parsers, the snippet is added to the appropriate parsing routine

25

Ad-Hoc Analysis

- In a **bottom-up** shift-reduce parsers, the **actions** are performed each time the parser performs a **reduction**.

26

LR(1) Skeleton Parser

```
stack.push(dummy); stack.push(0);
done = false; token = scanner.next();
while (!done) {
    s = stack.top();
    if( Action[s,token] == "reduce A→β") {
        stack.pop(2×|β|);
        s = stack.top();
        stack.push(A);
        stack.push(Goto[s,A]);
    }
    else if( Action[s,token] == "shift i"){
        stack.push(token); stack.push(i);
        token = scanner.next();
    }
}
```

27

LR(1) Skeleton Parser

```
stack.push(dummy); stack.push(0);
done = false; token = scanner.next();
while (!done) {
    s = stack.top();
    if( Action[s,token] == "reduce A→β") {
        stack.pop(2×|β|);
        s = stack.top();
        stack.push(A);
        stack.push(Goto[s,A]);
    }
    else if( Action[s,token] == "shift i"){
        stack.push(token); stack.push(i);
        token = scanner.next();
    }
}
```

28

LR(1) Skeleton Parser

```
if( Action[s,token] == "reduce A→β") {
    stack.pop(2×|β|);
    invoke the code snippet
    s = stack.top();
    stack.push(A);
    stack.push(Goto[s,A]);
}
```

29

<i>Productions</i>	<i>Code snippet</i>

Compiler Construction

Sohail Aslam

Lecture 33

Implementing Ad-Hoc Scheme

- Parser needs a **mechanism** to **pass values** of attributes from definitions in one snippet to uses in another

2

Implementing Ad-Hoc Scheme

- We will adopt **notation** used by **YACC** for snippets and passing values

3

Implementing Ad-Hoc Scheme

- Recall that the skeleton LR(1) parser stored **two** values on the stack $\langle \text{symbol}, \text{state} \rangle$
- We can replace this with triples $\langle \text{value}, \text{symbol}, \text{state} \rangle$

4

Implementing Ad-Hoc Scheme

- Recall that the skeleton LR(1) parser stored **two** values on the stack $\langle \text{symbol}, \text{state} \rangle$
- We can replace this with triples $\langle \text{value}, \text{symbol}, \text{state} \rangle$

5

Implementing Ad-Hoc Scheme

- On a **reduction** by $A \rightarrow \beta$, the parser pops $3 \times |\beta|$ items from the stack rather than $2 \times |\beta|$
- It pushes value along with the symbol

6

Implementing Ad-Hoc Scheme

- On a **reduction** by $A \rightarrow \beta$, the parser pops $3 \times |\beta|$ items from the stack rather than $2 \times |\beta|$
- It **pushes value** along with the symbol

7

YACC file for a calculator

```
%token NUMBER LPAREN RPAREN
%token PLUS MINUS TIMES DIVIDE
%%
expr : expr PLUS expr
     | expr MINUS expr
     | expr TIMES expr
     | expr DIVIDE expr
     | LPAREN expr RPAREN
     | MINUS expr
     | NUMBER
    ;
%%
```

8

```
%{
#include <iostream>
%}
%union {int val;}
%token NUMBER LPAREN RPAREN EQUAL
%token PLUS MINUS TIMES DIVIDE
/* associativity and precedence:
   in order of increasing precedence */
%nonassoc EQUAL
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS /* dummy token to use as
               precedence marker */
%type <val> NUMBER expr
%%
```

9

struct and union

```
struct rec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

sizeof v: 13 bytes,
sizeof w: 13 bytes

```
v.x = 10; v.y = 0.345; v.c = '#';  
w.x = 20; w.y = 24.05; w.c = '$';
```

10

struct and union

```
struct rec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

sizeof v: 13 bytes,
sizeof w: 13 bytes

```
v.x = 10; v.y = 0.345; v.c = '#';  
w.x = 20; w.y = 24.05; w.c = '$';
```

11

struct and union

```
struct rec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

sizeof v: 13 bytes,
sizeof w: 13 bytes

```
v.x = 10; v.y = 0.345; v.c = '#';  
w.x = 20; w.y = 24.05; w.c = '$';
```

12

struct and union

```
union urec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

sizeof v: 8 bytes,
sizeof w: 8 bytes

assign to one of the 3 fields
v.x, v.y, or v.c

13

struct and union

```
union urec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

sizeof v: 8 bytes,
sizeof w: 8 bytes

assign to one of the 3 fields
v.x, v.y, or v.c

14

struct and union

```
union urec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v, w;
```

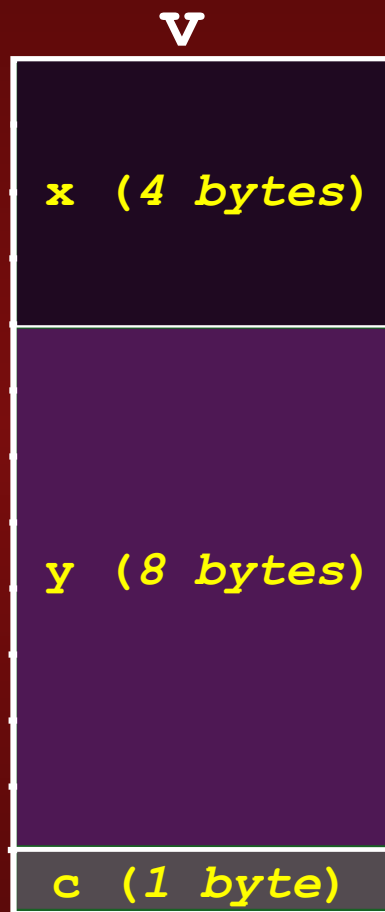
sizeof v: 8 bytes,
sizeof w: 8 bytes

assign to **one of** the 3 fields
v.x, v.y, or v.c

15

struct

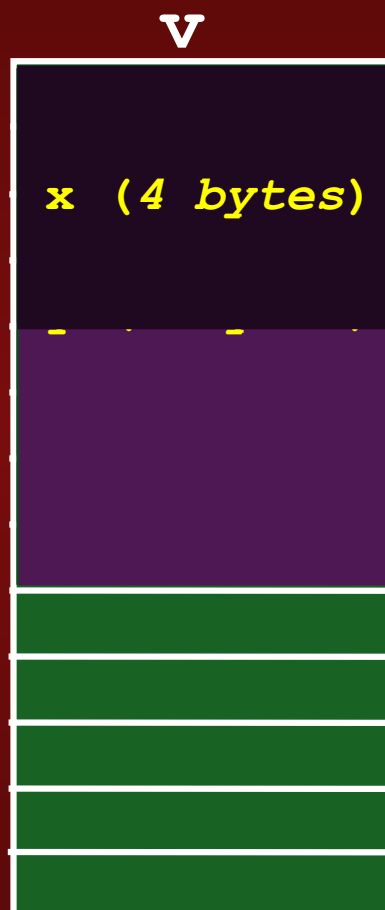
```
struct rec {  
    int x;  
    double y;  
    char c;  
} v;
```



16

union

```
union urec {  
    int x;  
    double y;  
    char c;  
} v;
```



17

struct and union

```
union urec {  
    int x;    // 4 bytes  
    double y; // 8 bytes  
    char c;   // 1 byte  
} v;
```

```
short int whichOne;
```

```
v.x = 10; whichOne = 1;  
v.y = 25.4; whichOne = 2;  
v.c = 'a'; whichOne = 3;
```

18

Compiler Construction

Sohail Aslam

Lecture 34

```
prog : expr { cout << $1 << endl; }
```

```
  ↑      ↑  
  $$     $1
```

```
expr : expr PLUS expr { $$ = $1 + $3; }  
      | expr MINUS expr { $$ = $1 - $3; }  
      | expr TIMES expr { $$ = $1 * $3; }  
      | expr DIVIDE expr { if($3)  
                           $$ = $1 / $3; }  
      | LPAREN expr RPAREN { $$ = $2; }  
      | MINUS expr          { $$ = -$2; }  
      | NUMBER              { $$ = $1; }  
      ;
```

%%

2

```
prog : expr { cout << $1 << endl; }  
      ;
```

```
  $$      $1      $2      $3
```

```
expr : expr PLUS expr { $$ = $1 + $3; }  
      | expr MINUS expr { $$ = $1 - $3; }  
      | expr TIMES expr { $$ = $1 * $3; }  
      | expr DIVIDE expr { if($3)  
                           $$ = $1 / $3; }  
      | LPAREN expr RPAREN { $$ = $2; }  
      | MINUS expr          { $$ = -$2; }  
      | NUMBER              { $$ = $1; }  
      ;
```

%%

3

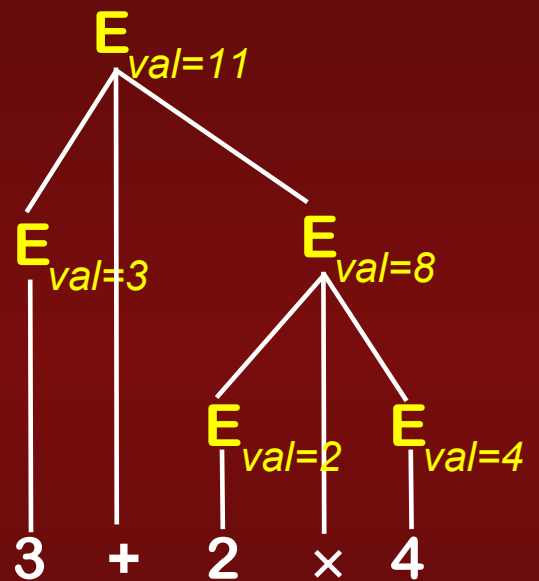
```
prog : expr { cout << $1 << endl; }
      ;
```

```
expr : expr PLUS expr { $$ = $1 + $3; }
     | expr MINUS expr { $$ = $1 - $3; }
     | expr TIMES expr { $$ = $1 * $3; }
     | expr DIVIDE expr { if($3)
                           $$ = $1 / $3; }
     | LPAREN expr RPAREN { $$ = $2; }
     | MINUS expr          { $$ = -$2; }
     | NUMBER              { $$ = $1; }
     ;
```

%%

4

→ 1	. 3 + 2 × 4	shift
→ 2	3 . + 2 × 4	reduce(r8)
→ 3	E . + 2 × 4	shift
→ 4	E + . 2 × 4	shift
→ 5	E + 2 . × 4	reduce(r8)
→ 6	E + E . × 4	shift
→ 7	E + E × . 4	shift
→ 8	E + E × 4 .	reduce(r8)
→ 9	E + E × E .	reduce(r4)
→ 10	E + E .	reduce(r2)
→ 11	E .	accept



5

Intermediate Representations (IR)

6

IR

- Compilers are organized as a series of **passes**
- This creates the need for an intermediate representation for the code being compiled

7

IR

- Compilers are organized as a series of **passes**
- This creates the need for an **intermediate representation** for the code being compiled

8

IR

- Compilers use some internal form— an **IR** —to represent the code being analysed and translated
- Many compilers use more than one IR during the course of compilation

9

IR

- Compilers use some internal form— an **IR** —to represent the code being analysed and translated
- Many compilers use **more than one** IR during the course of compilation

10

IR

- The IR must be expressive enough to **record** all of the useful **facts** that might be passed between passes of the compiler

11

IR

- During translation, the compiler **derives facts** that have no representation in the source code
- For example, the addresses of variables and procedures

12

IR

- During translation, the compiler **derives facts** that have no representation in the source code
- For example, the **addresses** of variables and procedures

13

IR

- Typically, the compiler **augments** the IR with a set of **tables** that record additional information
- These tables are considered part of the IR

14

IR

- Typically, the compiler **augments** the IR with a set of **tables** that record additional information
- These tables are considered part of the IR

15

IR

- **Selecting** an appropriate IR for a compiler project requires an understanding of both the **source language** and the **target machines** and the properties of the programs to be compiled

16

IR

- Thus, a **source-to-source** translator might keep its **internal** information in a **form** quite to **close** to the **source**

17

IR

- In contrast, a compiler that produces **assembly code** might use a form **close** to the target machine's **instruction set**

18

Compiler Construction

Sohail Aslam

Lecture 35

IR Taxonomy

IRs fall into **three** organizational categories

1. **Graphical** IRs encode the compiler's knowledge in a graph.

2

IR Taxonomy

2. **Linear** IRs resemble pseudocode for some abstract machine

3. Hybrid IRs combine elements of both graphical (structural) and linear IRs

3

IR Taxonomy

2. **Linear** IRs resemble pseudocode for some abstract machine

3. **Hybrid** IRs combine elements of both graphical (structural) and linear IRs

4

Graphical IRs

- **Parse trees** are graphs that represent **source-code form** of the program
- The structure of the tree corresponds to the syntax of the source code

5

Graphical IRs

- Parse trees are graphs that represent source-code form of the program
- The **structure** of the tree corresponds to the **syntax** of the source code

6

Graphical IRs

- **Parse trees** are used primarily in discussion of **parsing** and in **attribute grammar** systems where they are the primary IR
- In most other applications, compilers use one of the more concise alternatives

7

Graphical IRs

- Parse trees are used primarily in discussion of parsing and in attribute grammar systems where they are the primary IR
- In most **other applications**, compilers use one of the more concise **alternatives**

8

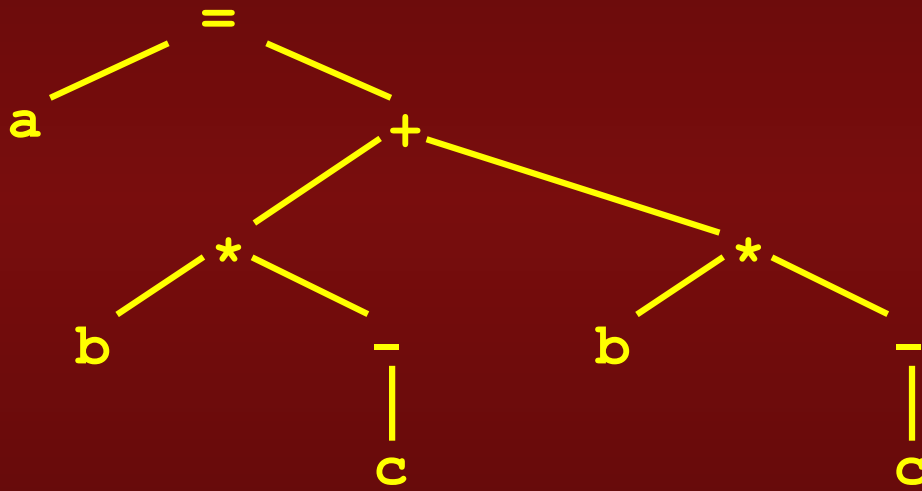
Graphical IRs

- **Abstract Syntax Trees** (AST) retains the essential structure of the parse tree but **eliminates** extraneous nodes

9

Graphical IRs

AST: $a = b * -c + b * -c$



10

Graphical IRs

ASTs have been used in many practical compiler systems

- Source-to-source systems
- automatic parallelization tools
- pretty-printing

11

Graphical IRs

ASTs have been used in many practical compiler systems

- Source-to-source systems
- automatic parallelization tools
- pretty-printing

12

Graphical IRs

ASTs have been used in many practical compiler systems

- Source-to-source systems
- automatic parallelization tools
- pretty-printing

13

Graphical IRs

ASTs have been used in many practical compiler systems

- Source-to-source systems
- automatic parallelization tools
- pretty-printing

14

Graphical IRs

- AST is more concise than a parse tree
- It faithfully retains the structure of the original source code
- Consider the AST for
$$x*2+x*2*y$$

15

Graphical IRs

- AST is more concise than a parse tree
- It faithfully retains the structure of the original source code

- Consider the AST for

$x^2 + x^2 * y$

16

Graphical IRs

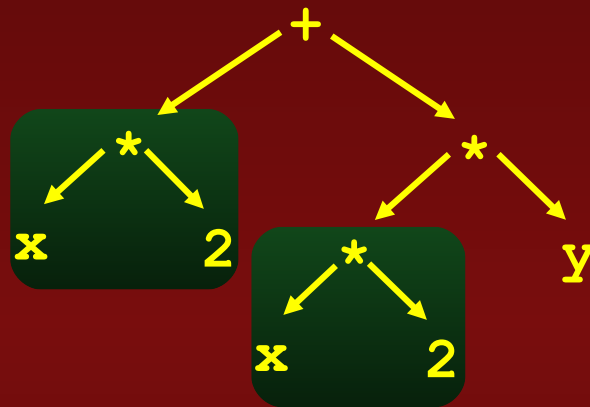
- AST is more concise than a parse tree
- It faithfully retains the structure of the original source code

- Consider the AST for

$x^2 + x^2 * y$

17

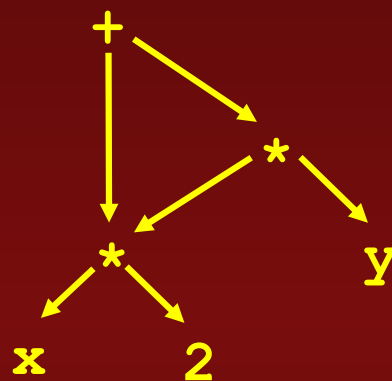
Graphical IRs



AST contains two distinct copies of $x * 2$

18

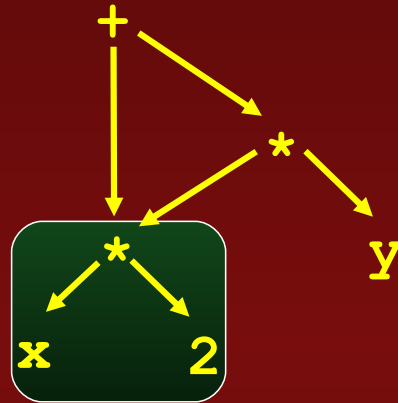
Graphical IRs



A **directed acyclic graph** (DAG) is a contraction of the AST that avoids duplication

19

Graphical IRs



If the **value** of **x** does **not change** between **uses** of **x*2**, the compiler can generate code that **evaluates** the subtree **once** and **uses** the result **twice**

20

Graphical IRs

- The task of **building AST** fits neatly into an ad hoc-syntax-directed translation scheme
- Assume that the compiler has routines **mknode** and **mkleaf** for creating tree nodes

21

Graphical IRs

- The task of building AST fits neatly into an ad hoc-syntax-directed translation scheme
- Assume that the compiler has routines **mknode** and **mkleaf** for creating tree nodes

22

<i>Production</i>	<i>Semantic Rule</i>

23

<i>Production</i>	<i>Semantic Rule (yacc)</i>

24

Intermediate Languages

- We will use another IR, called **three-address code**, for actual code generation
- The semantic rules for generating three-address code from high-level languages constructs are similar to those for AST.

25

Intermediate Languages

- We will use another IR, called **three-address code**, for actual code generation
- The semantic rules for generating three-address code for common **programming languages constructs** are similar to those for AST.

26

Linear IRs

- The alternative to graphical IR is a **linear IR**
- An assembly-language program is a form of linear code
- It consists of a sequence of instructions that execute in order of appearance

27

Linear IRs

- The alternative to graphical IR is a **linear IR**
- An **assembly-language** program is a form of linear code
- It consists of a sequence of instructions that execute in order of appearance

28

Linear IRs

- The alternative to graphical IR is a **linear IR**
- An **assembly-language** program is a form of linear code
- It consists of a **sequence of instructions** that execute in order of appearance

29

Linear IRs

Two linear IRs used in modern compilers are

- stack-machine code
- three-address code

30

Linear IRs

Linear IR for $x - 2 \times y$	
<i>stack-machine</i>	<i>three-address</i>
push 2	$t_1 \leftarrow 2$
push y	$t_2 \leftarrow y$
multiply	$t_3 \leftarrow t_1 \times t_2$
push x	$t_4 \leftarrow x$
subtract	$t_5 \leftarrow t_4 - t_3$

31

Linear IRs

Linear IR for $x - 2 \times y$	
<i>stack-machine</i>	<i>three-address</i>
push 2	$t_1 \leftarrow 2$
push y	$t_2 \leftarrow y$
multiply	$t_3 \leftarrow t_1 \times t_2$
push x	$t_4 \leftarrow x$
subtract	$t_5 \leftarrow t_4 - t_3$

32

Stack-Machine Code

- **Stack**-machine code is sometimes called **one-address code**
- It assumes the presence of an **operand** stack

Stack-Machine Code

- Most **operations** take their **operands** from the stack and **push results** back onto the stack
- Stack-machine code is **compact**; eliminates many names from IR
- This **shrinks** the program in IR form

34

Stack-Machine Code

- Most **operations** take their **operands** from the stack and **push results** back onto the stack
- Stack-machine code is **compact**; eliminates many names from IR
- This **shrinks** the program in IR form

35

Stack-Machine Code

- All results and arguments are **transitory** unless explicitly moved to memory
- Stack-machine code is simple to generate and execute

36

Stack-Machine Code

- All results and arguments are **transitory** unless explicitly moved to memory
- Stack-machine code is **simple** to **generate** and execute

37

Stack-Machine Code

- Smalltalk-80 and Java use bytecodes which are abstract stack-machine code
- The bytecode is either interpreted or translated into target machine code (JIT)

38

Stack-Machine Code

- Smalltalk-80 and Java use bytecodes which are abstract stack-machine code
- The bytecode is either interpreted or translated into target machine code (JIT)

39

Three-Address Code

Three-address code most operations have the form

$$\mathbf{x} \leftarrow \mathbf{y} \text{ } \mathbf{op} \text{ } \mathbf{z}$$

with an operator ($\mathbf{op}$), two operands ($\mathbf{y}$ and $\mathbf{z}$) and one result ($\mathbf{x}$)

40

Three-Address Code

- Some operators, such as an immediate load and a jump, will need fewer arguments

41

Compiler Construction

Sohail Aslam

Lecture 36

Three-Address Code

- Three-address code is **attractive** for several **reasons**:

Three-Address Code

- **absence** of **destructive** operators gives the compiler freedom to **reuse** names and values
- three-address code is reasonably compact: operations are 1 to 2 bytes; addresses are 4 bytes

3

Three-Address Code

- **absence** of **destructive** operators gives the compiler freedom to **reuse** names and values
- three-address code is reasonably **compact**: operations are 1 to 2 bytes; addresses are 4 bytes

4

Three-Address Code

- many modern processors implement three-address operations, a three-address code models their properties well

5

Syntax-directed Translation

- We now consider syntax-directed translation schemes using three-address code for various programming constructs

▪ We start with the assignment

6

Syntax-directed Translation

directed translation schemes using three-address code for various programming constructs

- We start with the **assignment statement**

7

<i>Production</i>	<i>translation scheme</i>
S $\rightarrow$ id = E	

8

<i>Production</i>	<i>translation scheme</i>
$E \rightarrow - E_1$	

9

Assignment Statement

- The translation scheme uses a **symbol table** for identifiers and temporaries
- Every time the parser encounters an identifier, it installs it in the symbol table.

Assignment Statement

- The translation scheme uses a **symbol table** for identifiers and temporaries
- Every time the parser encounters an **identifier**, it **installs** it in the **symbol table**.

11

Assignment Statement

- The symbol table can be **implemented** as a **hash table** or using some other **efficient** data structure for table.

12

Assignment Statement

- The routine **lookup (name)** checks if there an entry for the name in the symbol table
- If the **name** is found, the routine **returns** a **pointer** to entry.

13

Assignment Statement

- The routine **lookup (name)** checks if there an entry for the name in the symbol table
- If the **name** is found, the routine **returns** a **pointer** to entry.

14

Assignment Statement

- The routine **newtemp()** returns a new **temporary** in response to successive calls
- Temporaries can be **placed** in the **symbol table**

15

Assignment Statement

- The routine **emit()** generates a **three-address statement** which can either be held in **memory** or written to a **file**

16

Example

Here is the **bottom-up** parse of the assignment statement

$$a = b * -c + b * -c$$

and the syntax-directed **translation** into three-address code

17

Compiler Construction

Sohail Aslam

Lecture 37

$$a = b * -c + b * -c$$

<i>Parser action</i>	<i>attribute</i>	<i>code</i>
→ id=id * -id + id * -id		

2

<i>Parser action</i>	<i>attribute</i>	<i>code</i>
→		

3

$a = b * -c + b * -c$



$t1 = -c$

$t2 = b * t1$

$t3 = -c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

4

Representing Linear Codes

- Three-address codes are often implemented as a set of quadruples
- Each quadruple has four fields
 - an operator
 - two operands (or sources)
 - a destination

5

Representing Linear Codes

- Three-address codes are often implemented as a set of quadruples
- Each quadruple has four fields
 - an operator
 - two operands (or sources)
 - a destination

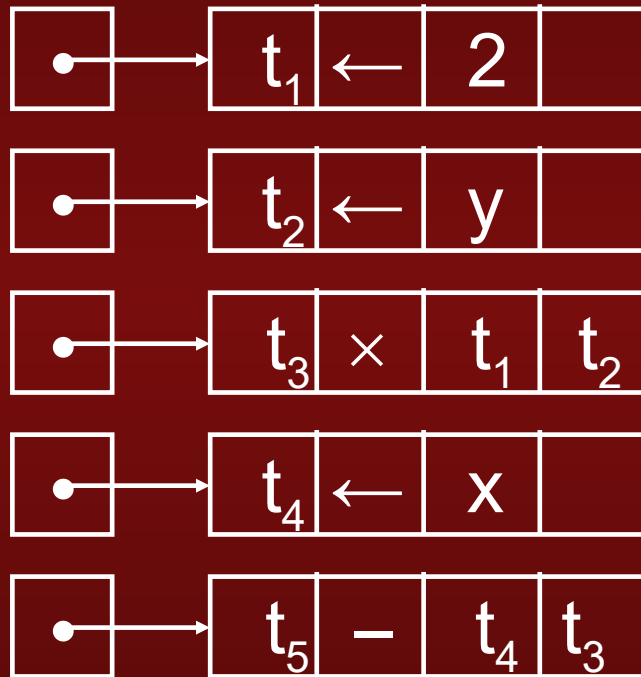
6

Simple Array of Quadruples

<i>Target</i>	<i>Op</i>	<i>Arg1</i>	<i>Arg2</i>
t_1	$\leftarrow$	2	
t_2	$\leftarrow$	y	
t_3	$\times$	t_1	t_2
t_4	$\leftarrow$	x	
t_5	$-$	t_4	t_3

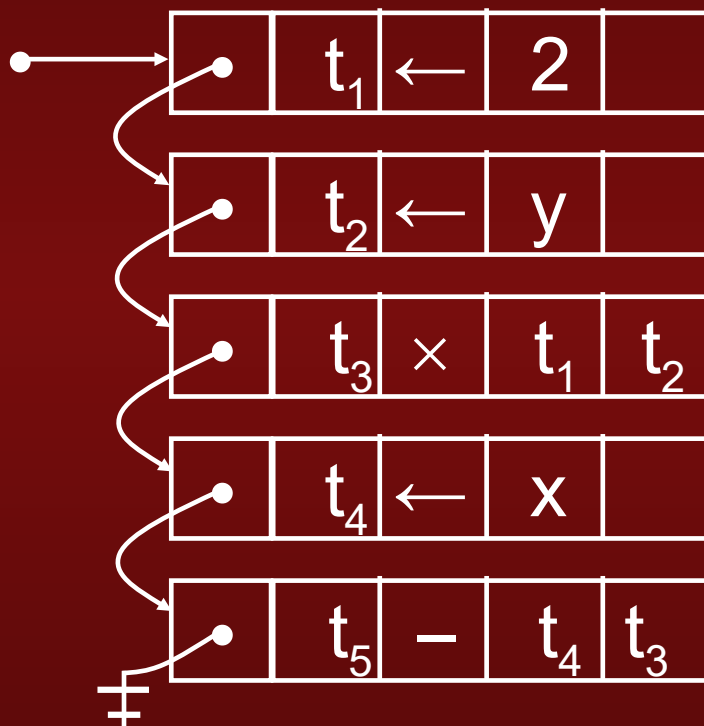
7

Array of Pointers to quads



8

Linked List of quads



9

Flow-of-Control Statements

$S \rightarrow$ if E then S_1
 | if E then S_1 else S_2
 | while E do S_1

where E is a boolean expression

10

Flow-of-Control Statements

- Consider the statement

```
if c < d then
    x = y + z
else
    x = y - z
```

- One possible 3-address code could be

11


```
if c < d then
    x = y + z
else
    x = y - z
```

```
    ◆
    if c < d goto L1
    ◆ goto L2
L1: x = y + z
    goto L3
L2: x = y - z
L3: nop
```

12

Flow-of-Control Statements

- we will assume that a three-address statement can be symbolically labeled
- the function newlabel() returns a new symbolic label each time it is called

13

Flow-of-Control Statements

- we will assume that a three-address statement can be symbolically labeled
- the function **newlabel()** returns a new symbolic label each time it is called

14

Compiler Construction

Sohail Aslam

Lecture 38

Three-Address Statement Types

- Prior to proceeding with flow-of-control construct, here are the types of three-Address statements that we will use

2

Three-Address Statement Types

Assignment statement

$x = y \text{ } op \text{ } z$

where ***op*** is a binary arithmetic or logical operation

3

Three-Address Statement Types

Assignment statement

$x = op\ y$

where **op** is a unary operation, e.g., unary minus, logical negation, shift operators

4

Three-Address Statement Types

Copy statement

$x = y$

where value of **y** is assigned to **x**

5

Three-Address Statement Types

Unconditional jump

goto L

The three-address statement with label **L** is executed next

6

Three-Address Statement Types

Conditional jump

if x *relop* y goto L

where ***relop*** is **<**, **=**, **>=**, etc.
If **x** stands in relation ***relop*** to **y**, execute statement with label **L**, next otherwise

7

Three-Address Statement Types

Indexed assignment

a) $\mathbf{x} = \mathbf{y}[\mathbf{i}]$

b) $\mathbf{x}[\mathbf{i}] = \mathbf{y}$

In **a)**, set **x** to value in location **i** memory units beyond location **y**.

8

Three-Address Statement Types

Indexed assignment

a) $\mathbf{x} = \mathbf{y}[\mathbf{i}]$

b) $\mathbf{x}[\mathbf{i}] = \mathbf{y}$

In **b)**, set contents of location **i** memory units beyond **x** to **y**.

9

Array of Quadruples

<i>Label</i>	<i>Target</i>	<i>Op</i>	<i>Arg1</i>	<i>Arg2</i>
	L1	if_lt	c	d
	L2	goto		
L1:	x	+	y	z
	L3	goto		
L2:	x	−	y	z
L3:		nop		

10

Flow-of-Control Statements

- We associate with a boolean expression **E** two labels; **E.true** and **E.false**
- The control flows to **E.true** if the expression evaluates to true, to **E.false** otherwise

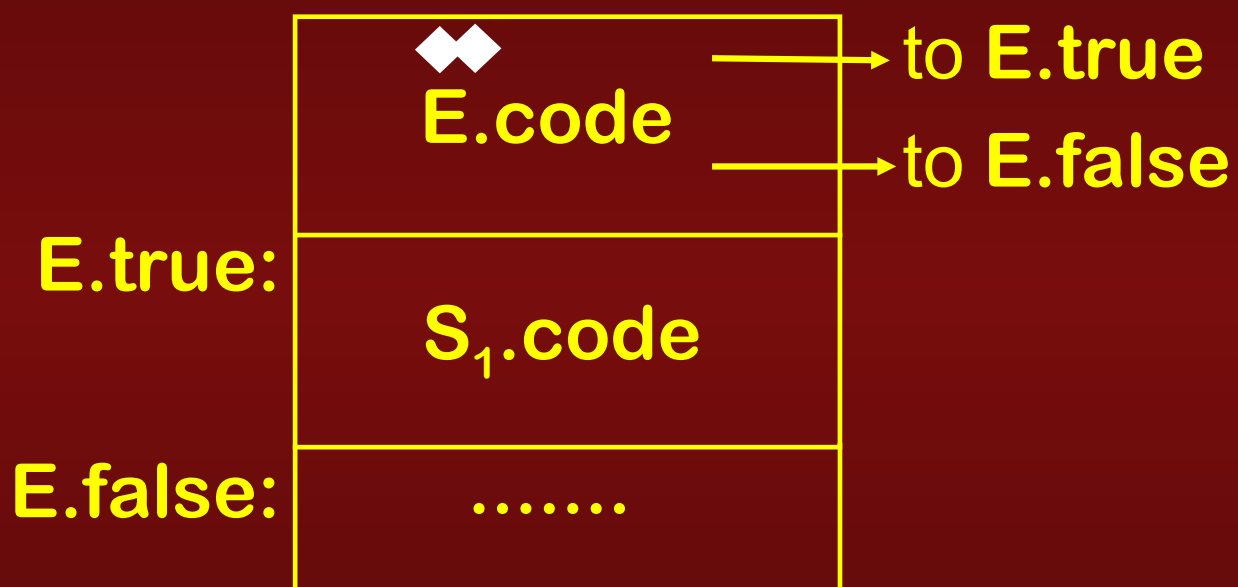
11

Flow-of-Control Statements

- We associate with a boolean expression **E** two labels; **E.true** and **E.false**
- The control flows to **E.true** if the expression evaluates to true, to **E.false** otherwise

12

$S \rightarrow \text{if } E \text{ then } S_1$



13

Syntax-directed translation for $S \rightarrow \text{if } E \text{ then } S_1$

→ **E.true = newlabel()**
E.false = S.next ← inherited attribute
S₁.next = S.next
S.code = E.code ||
gen(E.true ':') ||
S₁.code

14

Flow-of-Control Statements

- Suppose **E** is **a < b**. **E.code** is
if a < b goto E.true
goto E.false
- We will discuss semantic rules for boolean expressions shortly

15

Flow-of-Control Statements

Suppose E is $a < b$. E.code is

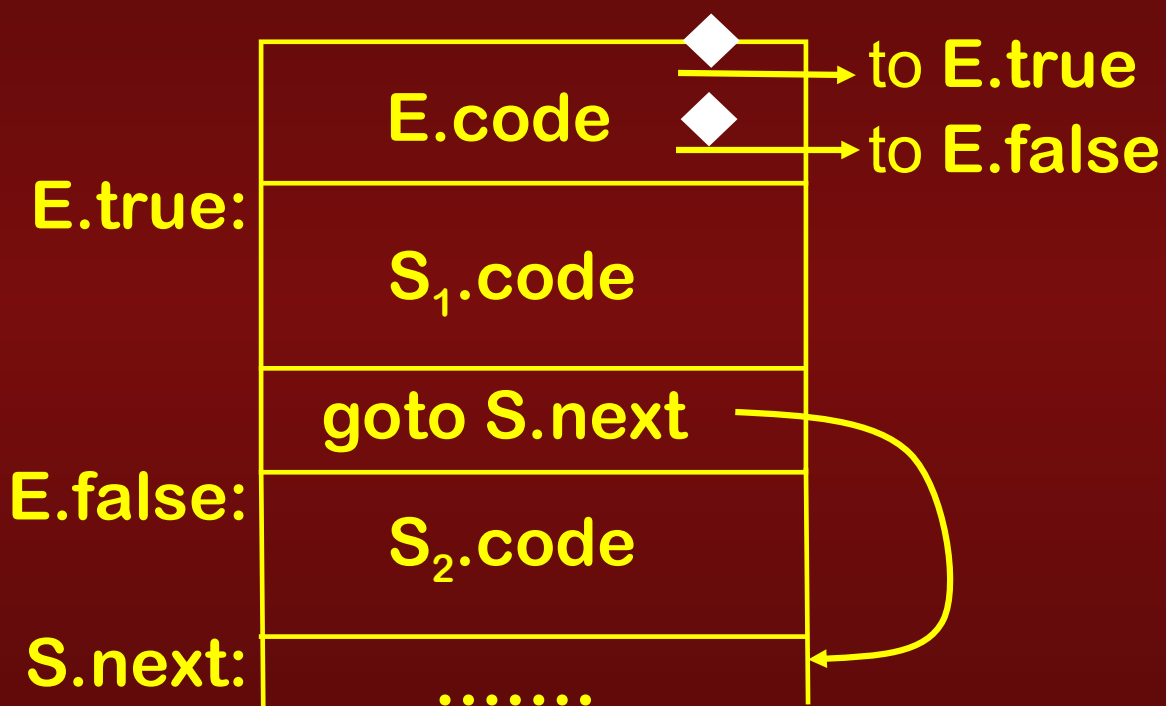
if $a < b$ goto E.true

goto E.false

- We will discuss semantic rules for **boolean expressions** shortly

16

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$



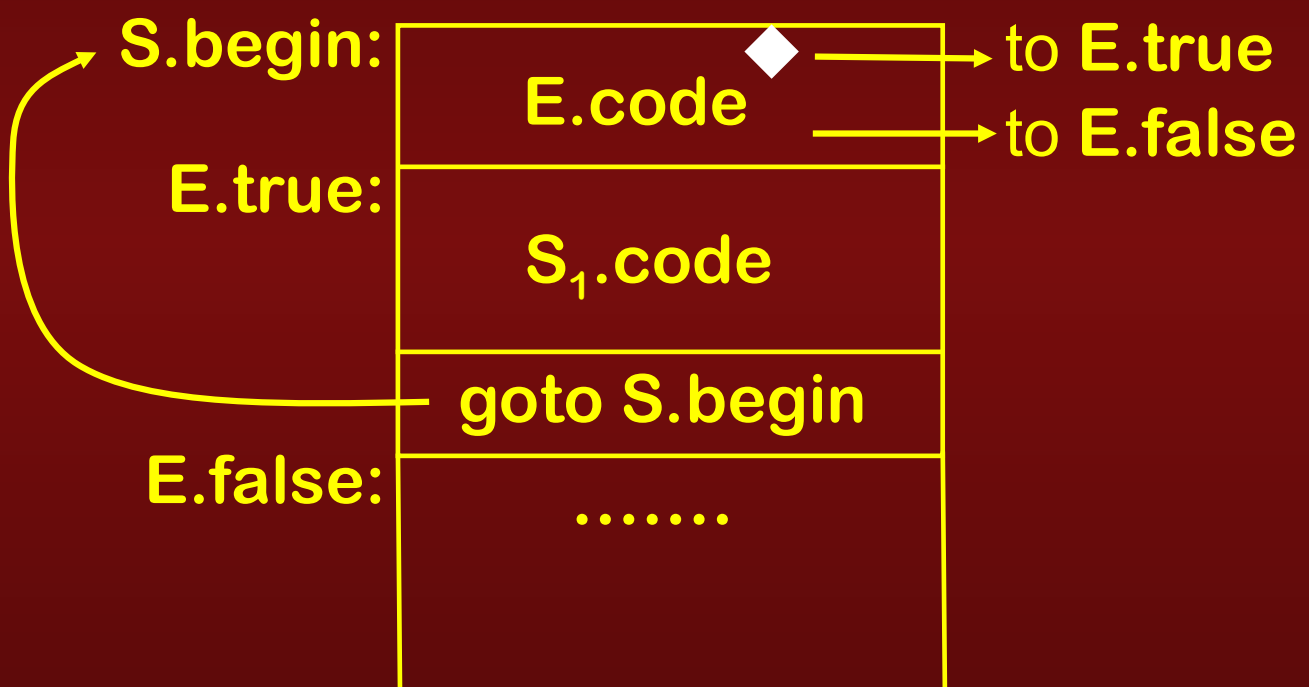
17

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

→ $E.\text{true} = \text{newlabel}()$
 $E.\text{false} = \text{newlabel}()$
 $S_1.\text{next} = S.\text{next}$ ← inherited attribute
 $S_2.\text{next} = S.\text{next}$ ←
 $S.\text{code} = E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel$
 $S_1.\text{code} \parallel$
 $\text{gen}(\text{'goto' } S.\text{next}) \parallel$
 $\text{gen}(E.\text{false} ':') \parallel$
 $S_2.\text{code}$

18

$\text{while } E \text{ do } S_1$



19

$S \rightarrow \text{while } E \text{ do } S_1$

```
→ S.begin = newlabel()
   E.true = newlabel()
   E.false = S.next
   S1.next = S.begin
   S.code = gen(S.begin ':') ||
            E.code ||
            gen(E.true ':') ||
            S1.code ||
            gen('goto' S.begin)
```

20

Compiler Construction

Sohail Aslam

Lecture 39

Boolean Expressions

In programming languages, **boolean expressions** have two primary purposes:

- compute logical values
x = a < b && d > e
- conditional expressions in flow-of-control statements

2

Boolean Expressions

In programming languages, **boolean expressions** have two primary purposes:

- compute logical values
x = a < b && d > e
- conditional expressions in flow-of-control statements

3

Boolean Expressions

Consider the grammar

$E \rightarrow$

- $E \text{ or } E$
- $E \text{ and } E$
- $\text{not } E$
- (E)
- id rel op id
- true
- false

4

Boolean Expressions

- We will implement the boolean expression by **flow of control method**
- We represent the value of a boolean expression by a position reached in the program

5

Boolean Expressions

- We will implement the boolean expression by **flow of control method**
- I.e., representing the **value** of a boolean expression by a **position reached** in the program

6

Boolean Expressions

$E \rightarrow id_1 \text{ relop } id_2$

E.code =

**gen('if' id₁ relop id₂ 'goto' E.true)
|| gen('goto' E.false)**

7

Boolean Expressions

$E \rightarrow \text{true}$

$E.\text{code} = \text{gen}(\text{'goto' } E.\text{true})$

$E \rightarrow \text{false}$

$E.\text{code} = \text{gen}(\text{'goto' } E.\text{false})$

8

Boolean Expressions

$E \rightarrow E_1 \text{ or } E_2$

→ $E_1.\text{true} = E.\text{true}$
 $E_1.\text{false} = \text{newlabel}()$
 $E_2.\text{true} = E.\text{true}$
 $E_2.\text{false} = E.\text{false}$
 $E.\text{code} = E_1.\text{code} ||$
 $\quad \text{gen}(E_1.\text{false} \text{'::'}) ||$
 $\quad E_2.\text{code}$



9

Boolean Expressions

$E \rightarrow E_1 \text{ and } E_2$

```
→ E1.true = newlabel()  
  E1.false = E.false  
  E2.true = E.true  
  E2.false = E.false  
  E.code = E1.code ||  
    gen(E1.true ':') ||  
    E2.code
```

10

Boolean Expressions

$E \rightarrow \text{not } E_1$

```
E1.true = E.false  
E1.false = E.true  
E.code = E1.code
```

11

Boolean Expressions

E $\rightarrow$ (E₁)

$E_1.\text{true} = E.\text{true}$

$E_1.\text{false} = E.\text{false}$

$E.\text{code} = E_1.\text{code}$

12

Boolean Expressions

- Consider the expression

$a < b \text{ or } c < d \text{ and } e < f$

- Suppose the true and false exits for the entire expression

13

Boolean Expressions

- Consider the expression
 $a < b \text{ or } c < d \text{ and } e < f$
- Suppose the **true** and **false** exits for the entire expression are **Ltrue** and **Lfalse**

14

```
if a < b or c < d and e < f
```

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

15

Consider the while statement

```
while a < b
    if c < d then
        x = y + z
    else
        x = y - z
```

16

```
while a < b
    if c < d then
        x = y + z
    else
        x = y - z
```

```
L1:  if a < b goto L2
      goto Lnext
```

```
L2:  if c < d goto L3
      goto L4
```

```
L3:  t1 = y + z
      x = t1
      goto L1
```

```
L4:  t2 = y - z
      x = t2
      goto L1
```

```
Lnext:  nop
```

17

Implementation

- The easiest way to implement syntax-directed definitions is to use **two** passes
 - construct a syntax tree for the input
 - walk the tree in depth-first order

18

Boolean Expressions

- The problem in generating three-address code in **one pass** is that we may not know the labels that the control must go to when we generate jump statements

19

Boolean Expressions

- However, by using a technique called **back-patching**, we can generate code in **one pass**.

20

Boolean Expressions

- we will generate the **jumps** with **targets** temporarily left unspecified

21

Boolean Expressions

- Each such statement will be put on a **list** of **goto** statements
- We will **fill** the labels when the proper label can be determined (backpatch)

22

Boolean Expressions

- Each such statement will be put on a **list** of **goto** statements
- We will **fill** the labels when the proper label can be **determined** (backpatch)

23

Backpatching

- Assume that the quadruples are put into an **array**
- **Labels** will be **indices** into this **array**
- To manipulate list of labels, we will use **three** functions:

24

Backpatching

1. **makelist(i)**

creates and returns a **new list** containing only **i**, the index of quadruple

25

Backpatching

2. **merge(p_1 , p_2)**

concatenates lists pointed to by **p_1** and **p_2** and returns the concatenated list

26

Backpatching

3. **backpatch(p , i)**

inserts **i** as the **target** label for **each** of the **goto** statements on list pointed to by **p**

27

Boolean Expressions

- We now construct a translation scheme suitable for producing quads (IR) for **boolean expressions** during **bottom-up** parsing

28

The grammar we use is

$E \rightarrow$ E_1 or $M E_2$
 | E_1 and $M E_2$
 | not E_1
 | (E_1)
 | id_1 **relop** id_2
 | true
 | false

$M \rightarrow \epsilon$

M is the marker non-terminal

M.quad records the number of **first** statement of E_2

29

Boolean Expressions

- We will associate synthesized attributes **truelist** and **falselist** with the nonterminal **E**
- **Incomplete** jumps will be placed on these list

30

Boolean Expressions

- We associate the semantic action

{ M.quad = nextquad() }

with the production **M** $\rightarrow \epsilon$

31

Boolean Expressions

- The function **nextquad()** returns the index of the next quadruple to follow

32

Boolean Expressions: AND

$E \rightarrow E_1 \text{ and } M \ E_2$

{

 backpatch(**E_1 .truelist**, **M .quad**);

E .truelist = **E_2 .truelist**;

E .falselist =

 merge(**E_1 .falselist**,

E_2 .falselist);

}

Let us look at the mechanics

33

Boolean Expressions



$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```

If E_1 is false, E is also false

34

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```



so the statements on E_1 .falselist become
part of E .falselist

35

Boolean Expressions



$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```

If E_1 is true, we must test E_2

36

Compiler Construction

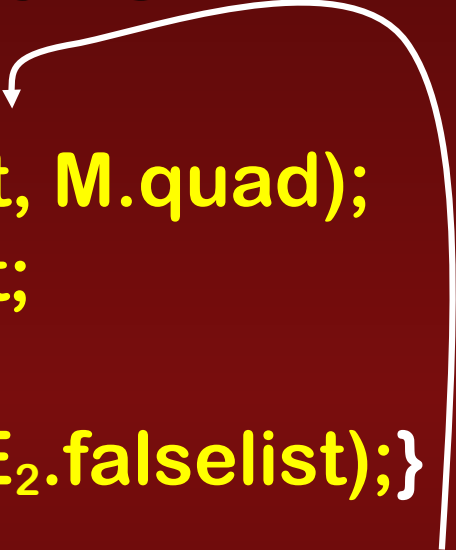
Sohail Aslam

Lecture 40

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```



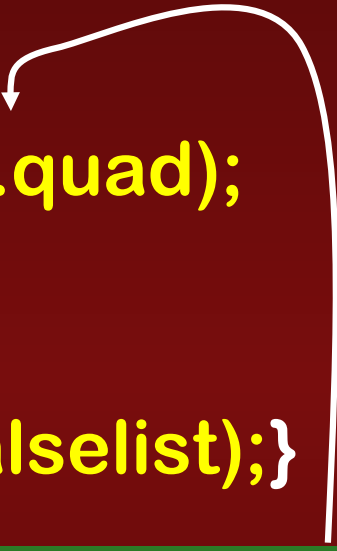
So the target for E_1 .truelist must be the beginning of code generated for E_2

38

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```



This target is obtained using the marker nonterminal M .

39

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

$\diamond$
{ backpatch(E_1 .truelist, M.quad);
 E .truelist = E_2 .truelist;
 E .falselist =
 merge(E_1 .falselist, E_2 .falselist);}

M.quad records the number of the first statement of **E_2 .code**.

40

Boolean Expressions: OR

$E \rightarrow E_1 \text{ or } M \ E_2$

$\diamond$
{
 backpatch(E_1 .falselist, M.quad);
 E .truelist =
 merge(E_1 .truelist, E_2 .truelist);
 E .falselist = E_2 .falselist;
}

If **E_1** is false, need to test **E_2**

41

Boolean Expressions

$E \rightarrow \text{not } E_1$

{

◆ $E.\text{truelist} = E_1.\text{falselist};$
 $E.\text{falselist} = E_1.\text{truelist};$

}

42

Boolean Expressions

$E \rightarrow (E_1)$

{

◆ $E.\text{truelist} = E_1.\text{truelist};$
 $E.\text{falselist} = E_1.\text{falselist};$

}

43

Boolean Expressions

$E \rightarrow id_1 \text{ relop } id_2$

```
{  
◆ E.truelist = makelist(nextquad());  
  E.falselist = makelist(nextquad()+1);  
  
  emit('if' id1 relop id2 'goto _');  
  emit('goto _' );  
}
```

44

Boolean Expressions

$E \rightarrow \text{true}$

```
{  
  E.truelist = makelist(nextquad());  
  emit('goto _' );  
}
```

45

Boolean Expressions

$E \rightarrow \text{false}$

```
{  
    E.falselist = makelist(nextquad());  
    emit('goto _' );  
}
```

46

Boolean Expressions

$M \rightarrow \varepsilon$

```
{  
    M.quad = nextquad();  
}
```

47

Backpatching

- consider again, the boolean expression

$a < b$ or $c < d$ and $e < f$

- We carry out a bottom-up parse

48

Backpatching

- In response to reduction of **$a < b$** to **E** , the two quadruples

```
100: if a < b goto _  
101: goto _
```

are generated

49

Recall

$E \rightarrow id_1 \text{ relop } id_2$

```
{  
◆ E.truelist = makelist(nextquad());  
  E.falselist = makelist(nextquad()+1);  
  
  emit('if' id1 relop id2 'goto _');  
  emit('goto _');  
}
```

View this in a parse tree

50

E.t = {100}

E.f = {101}



a < b or c < d and e < f

51

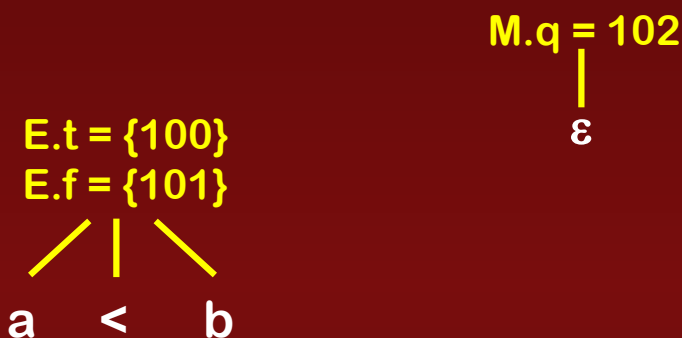
Backpatching

- The marker non-terminal **M** in the production

$$E \rightarrow E_1 \text{ or } M E_2$$

records the value of **nextquad** which at this time is **102**.

52



or $c < d$ and $e < f$

53

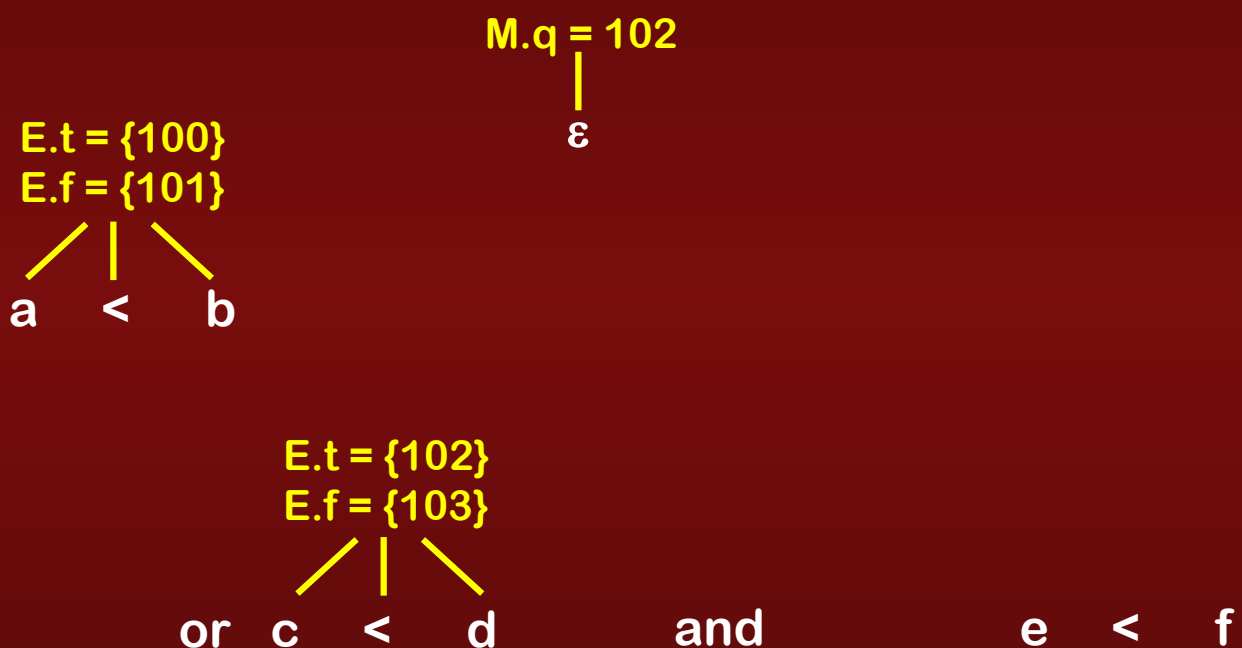
Backpatching

- The reduction of $c < d$ to E , the two quadruples

```
102: if c < d goto _  
103: goto _
```

are generated

54



55

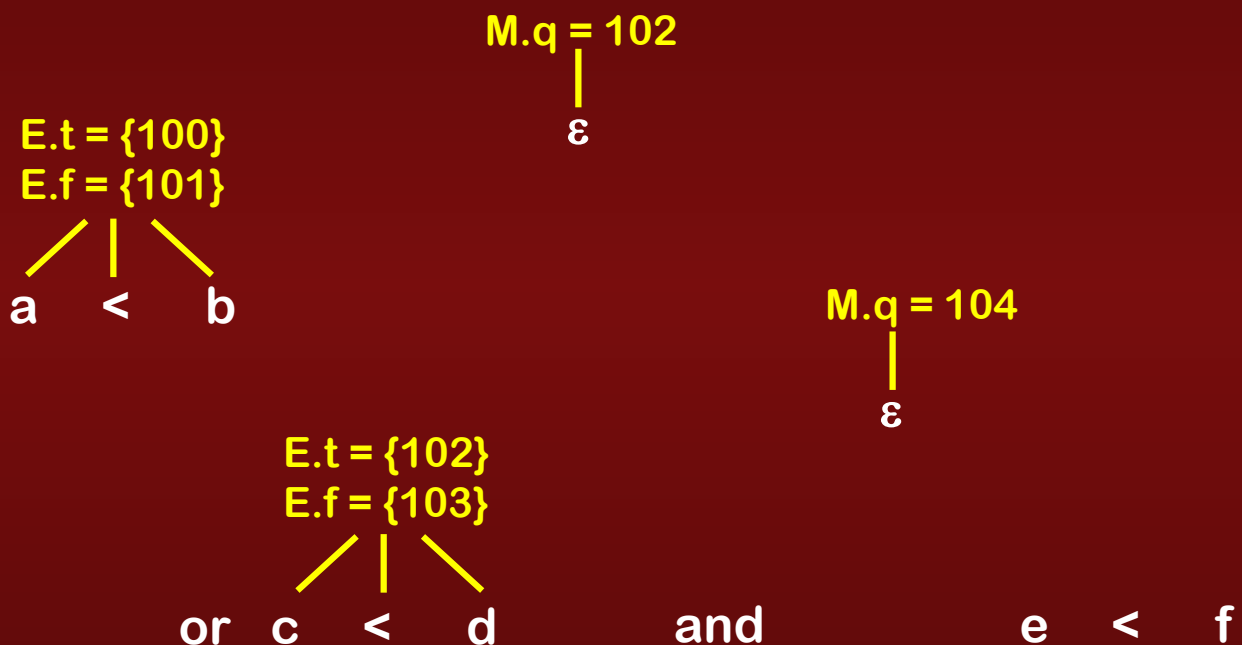
Backpatching

- The marker non-terminal **M** in the production

$$E \rightarrow E_1 \text{ and } M E_2$$

records the value of **nextquad** which at this time is **104**.

56



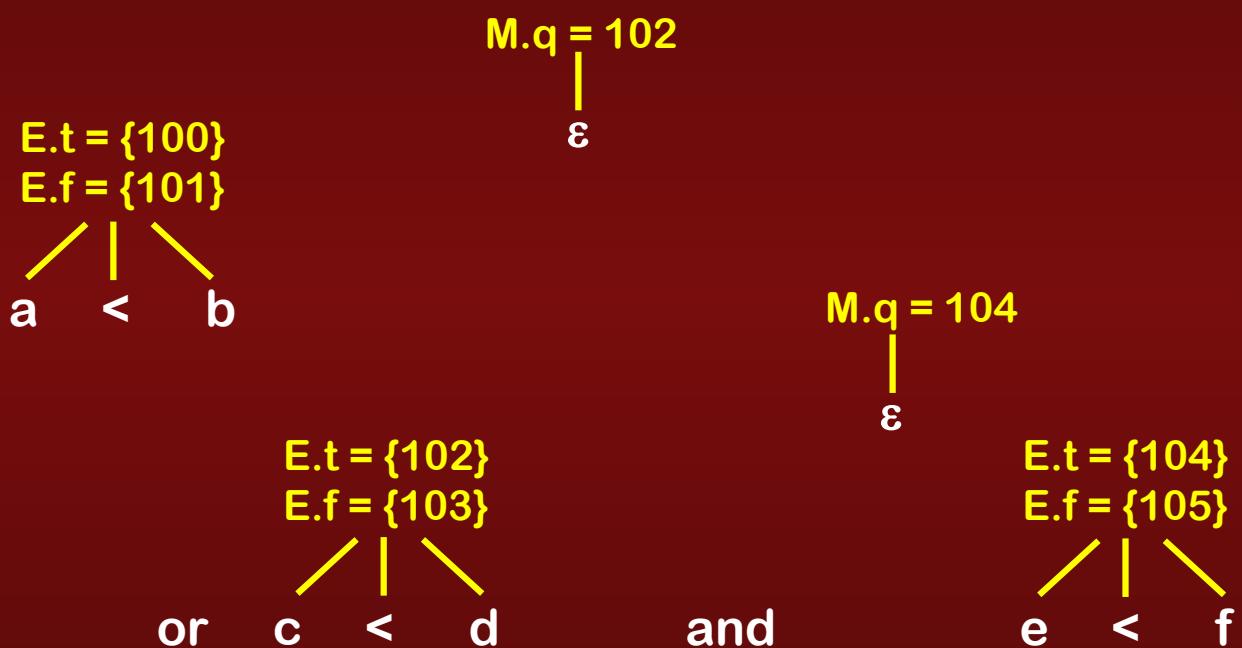
57

Backpatching

- Reducing $e < f$ to E , generates

```
104: if e < f goto _  
105: goto _
```

58



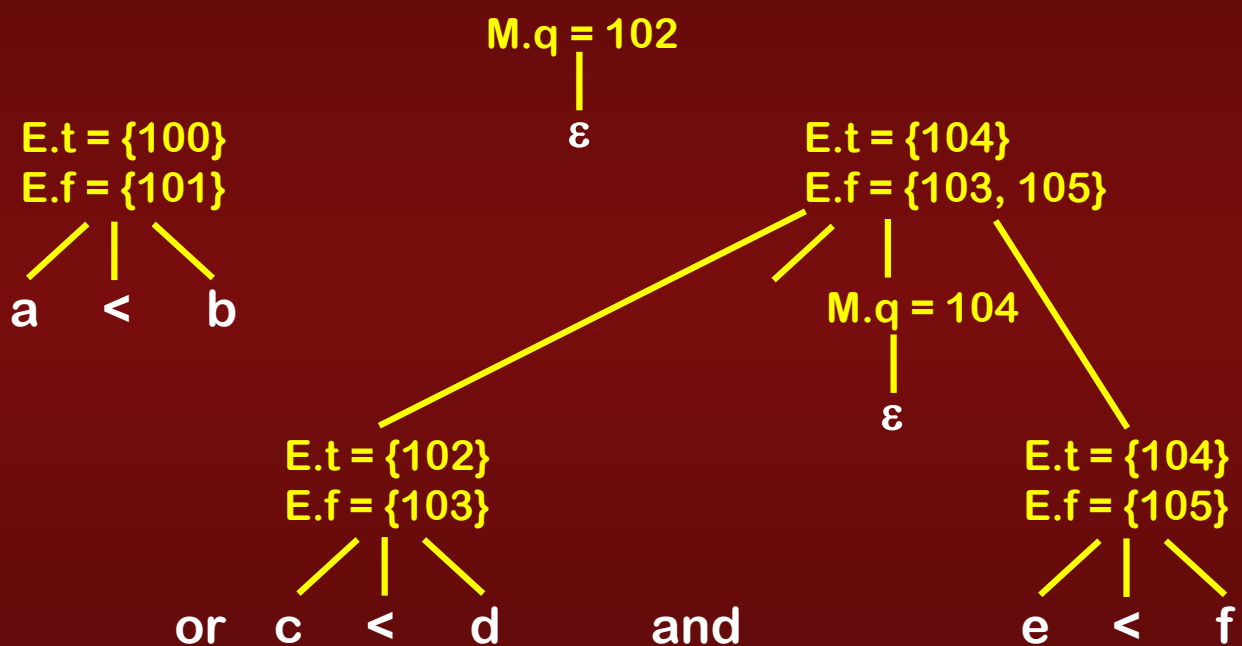
59

Backpatching

- We now reduce by the production

$$E \rightarrow E_1 \text{ and } M E_2$$

60



61

Semantic Actions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```

62

Backpatching

The semantic action calls

`backpatch({102}, 104)`

where `{102}` denotes `$E_1$ .truelist`
containing only 102

63

Backpatching

The six statements generated so far are thus

```
100:  if a < b goto _  
101:  goto _  
102:  if c < d goto _  
103:  goto _  
104:  if e < f goto _  
105:  goto _
```

64

Backpatching

The semantic action calls

backpatch({102},104)

which fills in **104** in statement **102**.

65

Backpatching

The call fills in 104 in statement 102

```
100: if a < b goto _  
101: goto _  
▶ 102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```

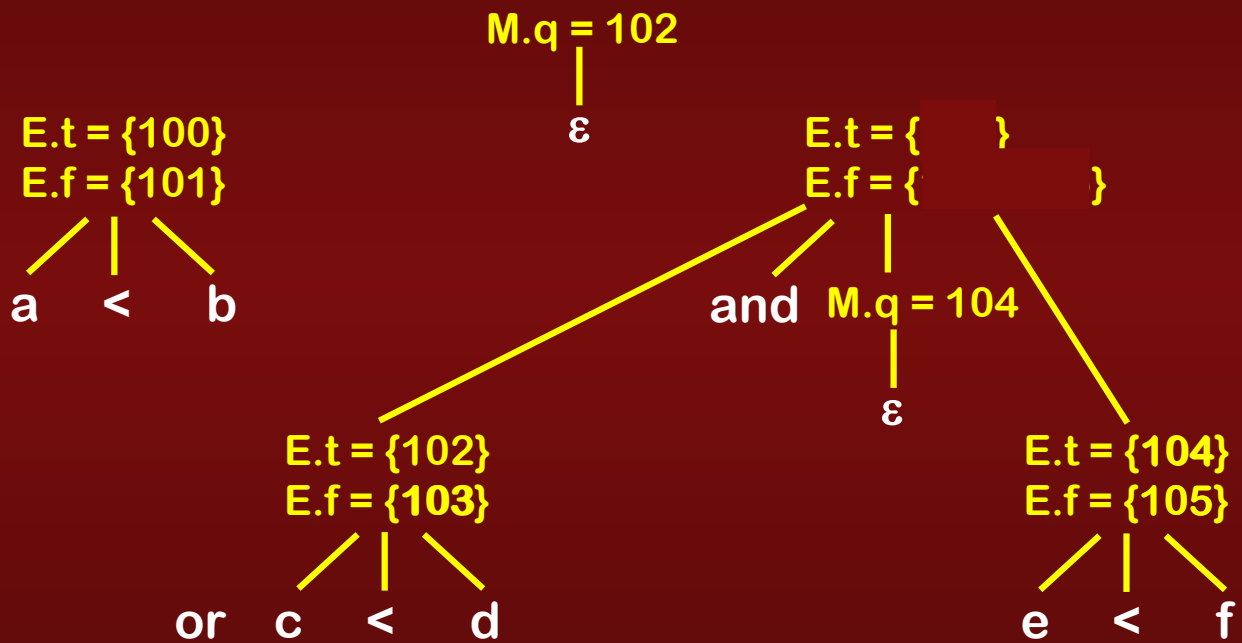
66

Semantic Actions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch(E1.truelist, M.quad);  
  E.truelist = E2.truelist;  
  E.falselist =  
    merge(E1.falselist, E2.falselist);}
```

67



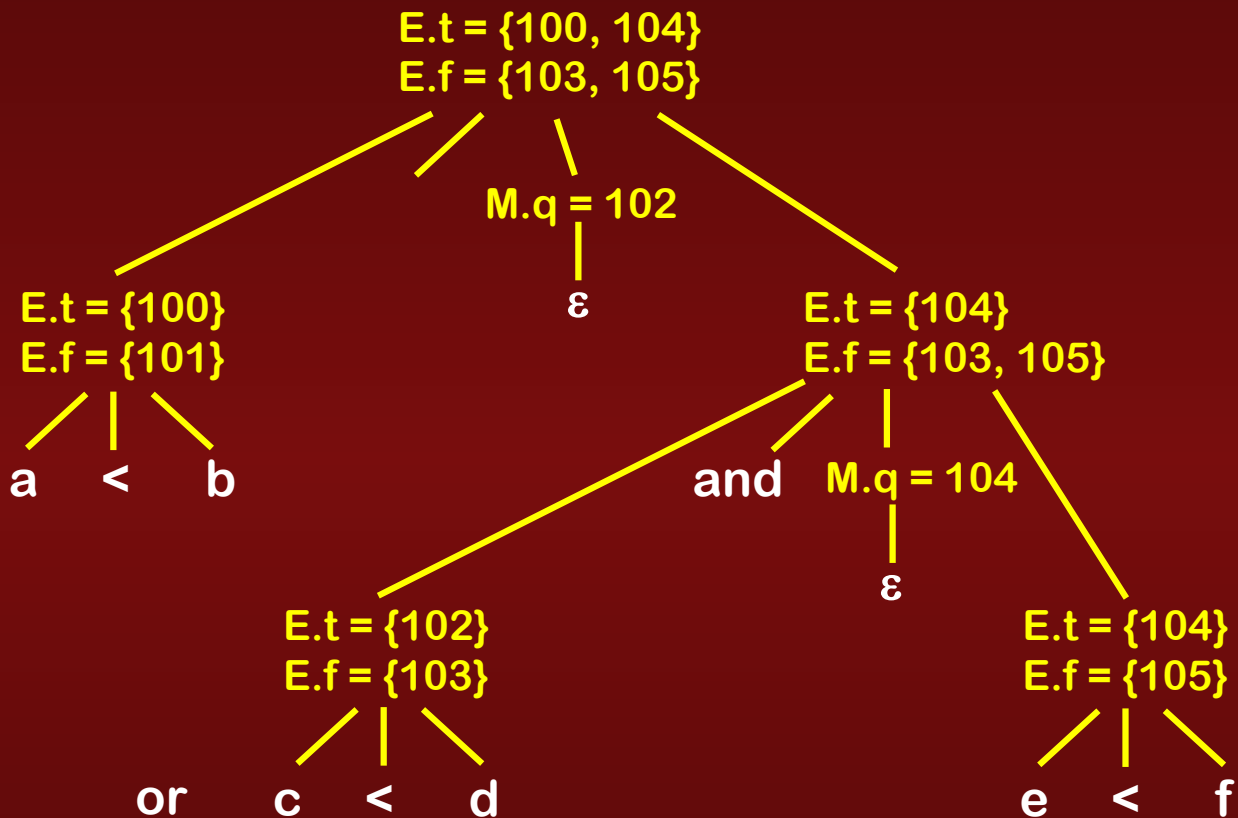
68

Backpatching

- We now reduce by the production

$$E \rightarrow E_1 \text{ or } M \ E_2$$

69



70

Semantic Actions

$E \rightarrow E_1 \text{ or } M E_2$

{

backpatch(E_1 .false_{list}, M .quad);

E .true_{list} =

merge(E_1 .true_{list}, E_2 .true_{list});

E .false_{list} = E_2 .false_{list};

}

71

Backpatching

The semantic action calls

backpatch({101},102)

which fills in **102** in statement **101**.

72

Backpatching

```
100:  if a < b goto _  
► 101:  goto 102  
102:  if c < d goto 104  
  
103:  goto _  
104:  if e < f goto _  
105:  goto _
```

73

Backpatching

```
100: if a < b goto _  
101: goto 102  
102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```



These instructions will have their targets filled later in the compilation

74

Semantic Actions

$E \rightarrow E_1 \text{ or } M \ E_2$

{

backpatch(E_1 .falselist, M.quad);

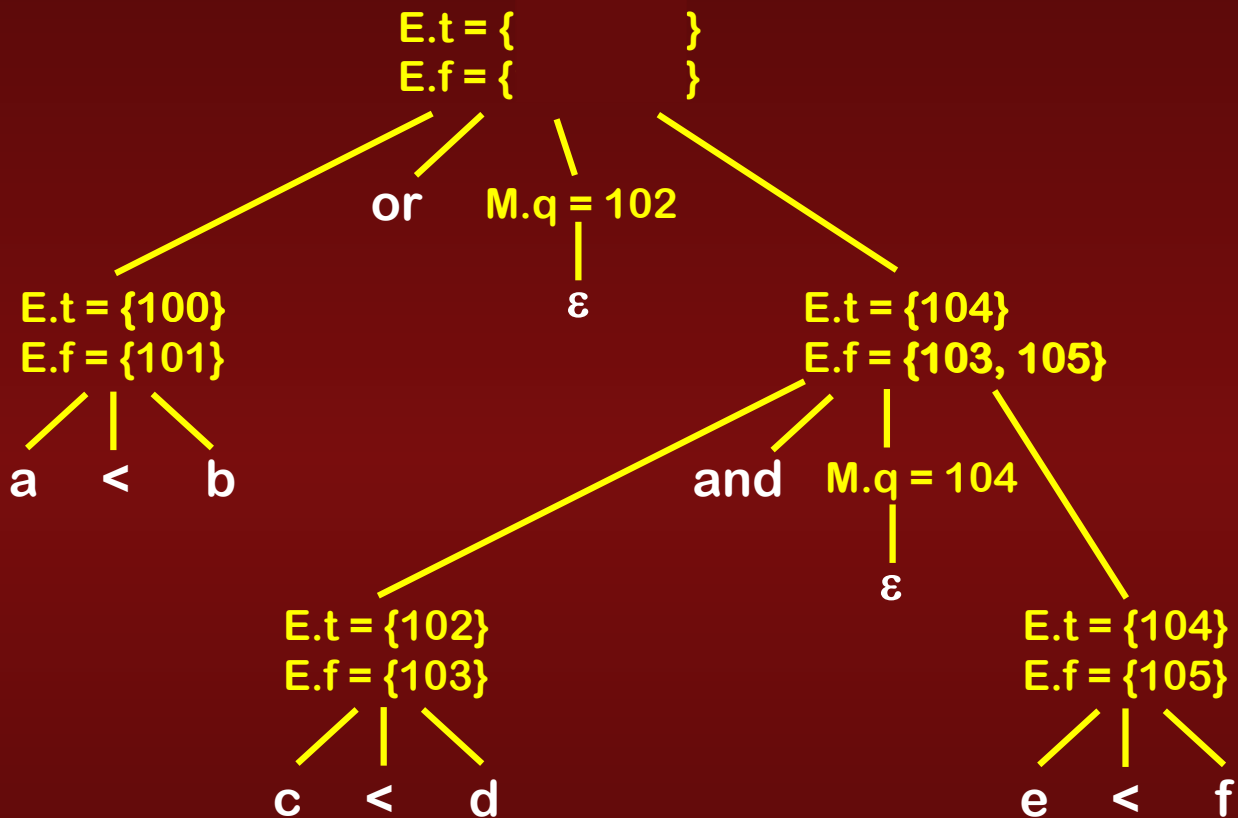
E .truelist =

merge(E_1 .truelist, E_2 .truelist);

E .falselist = E_2 .falselist;

}

75



76

Flow-of-Control Statements

- We now use **backpatching** to translate **flow-of-control** statements in **one pass**

we will use the same error handling procedures as before

77

Flow-of-Control Statements

- We now use **backpatching** to translate **flow-of-control** statements in **one pass**
- We will use the same list-handling procedures as before

78

Flow-of-Control Statements

S $\rightarrow$ **if E then S**

| **if E then S else S**

| **while E do S**

| **begin L end**

| **A**

L $\rightarrow$ **L ; S**

| **S**

S denotes a statement

L is a statementlist

A is assignment

79

Semantic Actions

$S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

{



backpatch(E.truelist, M_1 .quad);
backpatch(E.falselist, M_2 .quad);
 S .nextlist = merge(S_1 .nextlist,
merge(N .nextlist, S_2 .nextlist));

}

N is marker non-terminal to introduce
jump over code for S_2

80

Semantic Actions

$N \rightarrow \epsilon$

{

N .nextlist = makelist(nextQuad());
emit('goto_');

}

The attribute N .nextlist records the
quad number of the goto it generates

81

Semantic Actions

$M \rightarrow \varepsilon$

```
{  
    M.quad = nextQuad();  
}
```

82

Semantic Actions

$S \rightarrow \text{if } E \text{ then } M S_1$

```
{  
     backpatch(E.truelist, M.quad);  
    S.nextlist = merge(  
        E.falselist, S1.nextlist);  
}
```

If **E** is true, jump to **M.quad** which is start of code for **S₁**

83

Semantic Actions

$S \rightarrow \text{while } M_1 \text{ E do } M_2 S_1$

{

backpatch(S_1 .nextlist, M_1 .quad);
backpatch(E .truelist, M_2 .quad);
 S .nextlist = E .falselist;
emit('goto' M_1 .quad);

}

84

Semantic Actions

$S \rightarrow \text{begin } L \text{ end}$

{

S .nextlist = L .nextlist;

}

85

Semantic Actions

$S \rightarrow A$

```
{  
    S.nextlist = nil;  
}
```

initializes **S.nextlist** to an empty list

86

Semantic Actions

$S \rightarrow L_1 ; M S$

```
{  
      
    backpatch(L1.nextlist, M.quad);  
    L.nextlist = S.nextlist;  
}
```

Statement following **L₁** in order of execution is beginning of **S**

87

Semantic Actions

$L \rightarrow S$

{

$L.nextlist = S.nextlist;$

}

88

Example

if $a < b$ or $c < d$ and $e < f$ then $x = y + z$ else $x = y - z$

89

if E_1 or M E_2 then $x=y+z$ else $x=y-z$

if E then $x=y+z$ else $x=y-z$

90

→	100	if a < b goto _
	101	goto 102
	102	if c < d goto 104
→	103	goto _
→	104	if e < f goto _
→	105	goto _
	106	
	107	
	108	
	109	

91

if E then M_1 $x=y+z$ else $x=y-z$ $M_1 \rightarrow \varepsilon$

{ M_1 .quad = 106 }

if E then M_1 A else $x=y-z$
{ emit('x=y+z') }

$A \rightarrow x=y+z$

if E then M_1 S_1 else A
{ S_1 .nextlist = nil }

$S_1 \rightarrow A$

if E then M_1 S_1 N else $x=y-z$
{ N.nextlist = [107]
emit('goto _') }

$N \rightarrow \varepsilon$

92

	100	if a < b goto _
	101	goto 102
	102	if c < d goto 104
	103	goto _
	104	if e < f goto _
	105	goto _
→	106	x=y+z
→	107	goto _
	108	
	109	

93

100	if a < b goto 106
101	goto 102
102	if c < d goto 104
103	goto 108
104	if e < f goto 106
105	goto 108
106	x=y+z
107	goto _
108	x=y-z
109	

96

Semantic Actions in YACC

The syntax-directed translation statements can be conveniently specified in **YACC**

The **%union** will require more fields because the attributes vary

97

Semantic Actions in YACC

The actual mechanics will be covered in the **handout** for the syntax-directed translation phase of the **course project**

98

Compiler Construction

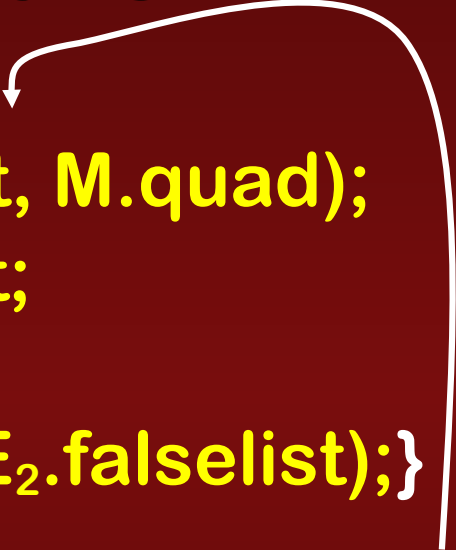
Sohail Aslam

Lecture 40

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```



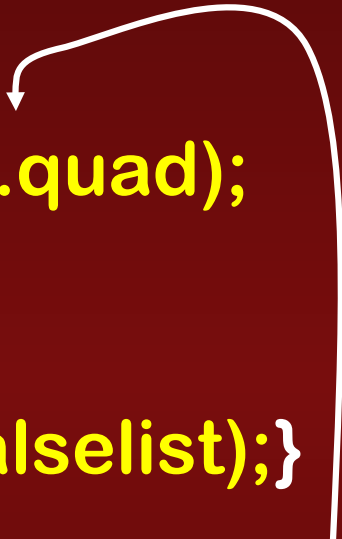
So the target for E_1 .truelist must be the beginning of code generated for E_2

2

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```



This target is obtained using the marker nonterminal M .

3

Boolean Expressions

$E \rightarrow E_1 \text{ and } M \ E_2$

$\diamond$
{ backpatch(E_1 .truelist, M.quad);
 E .truelist = E_2 .truelist;
 E .falselist =
 merge(E_1 .falselist, E_2 .falselist);}

M.quad records the number of the first statement of **E_2 .code**.

4

Boolean Expressions: OR

$E \rightarrow E_1 \text{ or } M \ E_2$

$\diamond$
{
 backpatch(E_1 .falselist, M.quad);
 E .truelist =
 merge(E_1 .truelist, E_2 .truelist);
 E .falselist = E_2 .falselist;
}

If E_1 is false, need to test E_2

5

Boolean Expressions

$E \rightarrow \text{not } E_1$

{

◆ $E.\text{truelist} = E_1.\text{falselist};$
 $E.\text{falselist} = E_1.\text{truelist};$

}

6

Boolean Expressions

$E \rightarrow (E_1)$

{

◆ $E.\text{truelist} = E_1.\text{truelist};$
 $E.\text{falselist} = E_1.\text{falselist};$

}

7

Boolean Expressions

$E \rightarrow id_1 \text{ relop } id_2$

```
{  
◆ E.truelist = makelist(nextquad());  
  E.falselist = makelist(nextquad()+1);  
  
  emit('if' id1 relop id2 'goto _');  
  emit('goto _' );  
}
```

8

Boolean Expressions

$E \rightarrow \text{true}$

```
{  
  E.truelist = makelist(nextquad());  
  emit('goto _' );  
}
```

9

Boolean Expressions

$E \rightarrow \text{false}$

```
{  
    E.falselist = makelist(nextquad());  
    emit('goto _' );  
}
```

10

Boolean Expressions

$M \rightarrow \epsilon$

```
{  
    M.quad = nextquad();  
}
```

11

Backpatching

- consider again, the boolean expression

$a < b$ or $c < d$ and $e < f$

- We carry out a bottom-up parse

12

Backpatching

- In response to reduction of **$a < b$** to **E** , the two quadruples

```
100: if a < b goto _  
101: goto _
```

are generated

13

Recall

$E \rightarrow id_1 \text{ relop } id_2$

```
{  
◆ E.truelist = makelist(nextquad());  
  E.falselist = makelist(nextquad()+1);  
  
  emit('if' id1 relop id2 'goto _');  
  emit('goto _');  
}
```

View this in a parse tree

14

E.t = {100}
E.f = {101}



a < b or c < d and e < f

15

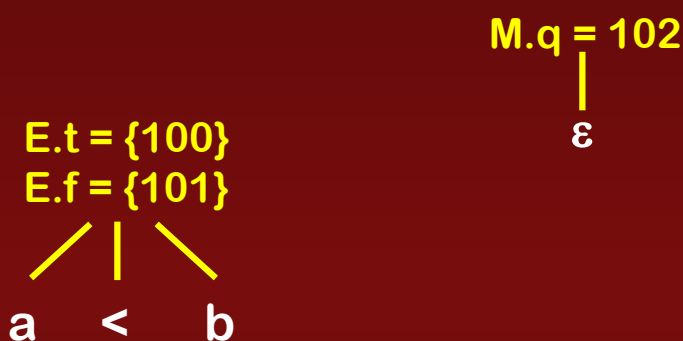
Backpatching

- The marker non-terminal **M** in the production

$$E \rightarrow E_1 \text{ or } M E_2$$

records the value of **nextquad** which at this time is **102**.

16



or $c < d$ and $e < f$

17

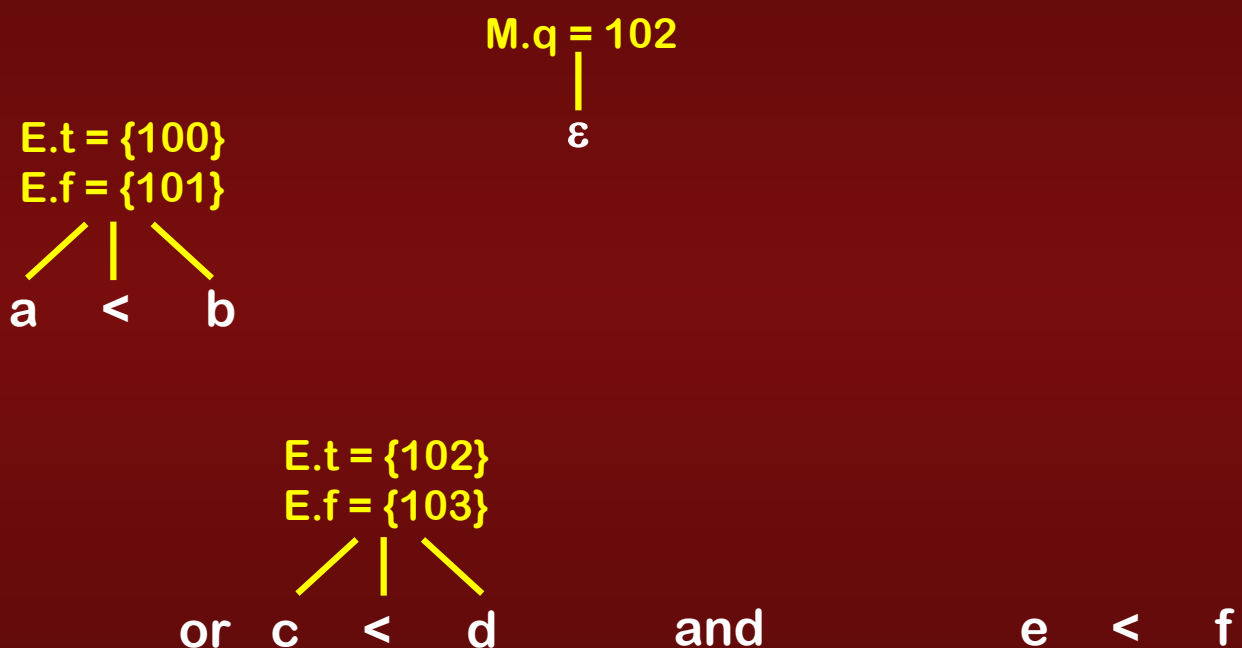
Backpatching

- The reduction of $c < d$ to E , the two quadruples

```
102: if c < d goto _  
103: goto _
```

are generated

18



19

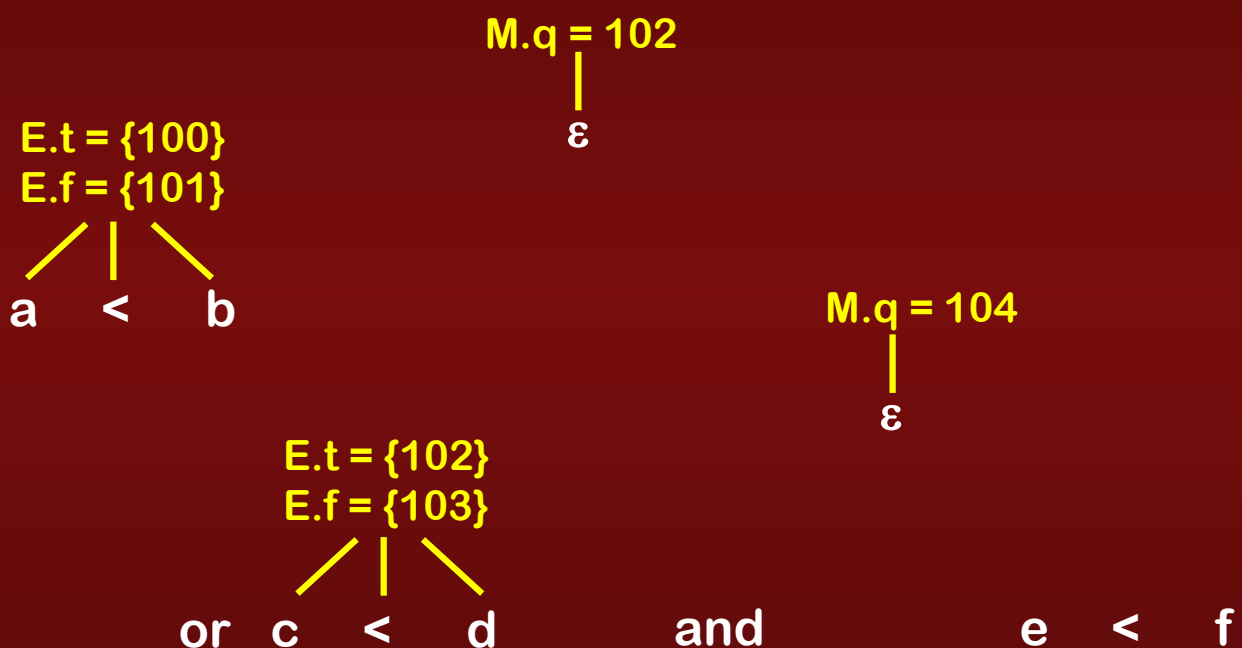
Backpatching

- The marker non-terminal **M** in the production

$$E \rightarrow E_1 \text{ and } M E_2$$

records the value of **nextquad** which at this time is **104**.

20



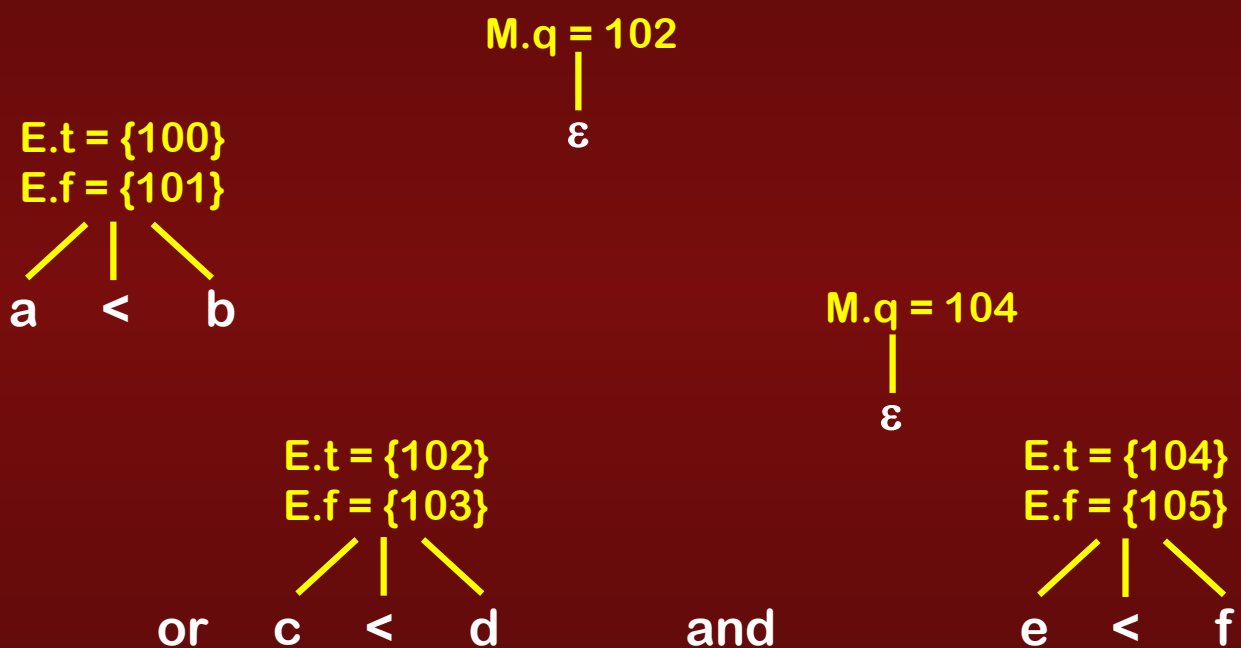
21

Backpatching

- Reducing $e < f$ to E , generates

```
104: if e < f goto _  
105: goto _
```

22



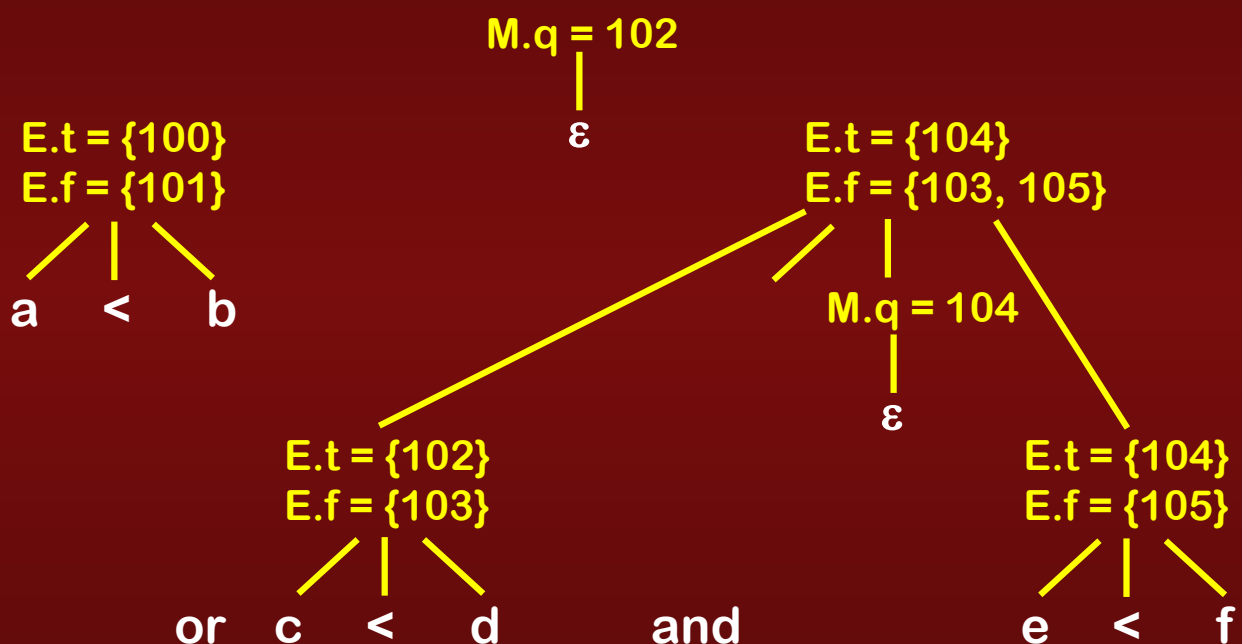
23

Backpatching

- We now reduce by the production

$$E \rightarrow E_1 \text{ and } M E_2$$

24



25

Semantic Actions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist =  $E_2$ .truelist;  
   $E$ .falselist =  
    merge( $E_1$ .falselist,  $E_2$ .falselist);}
```

26

Backpatching

The semantic action calls

`backpatch({102}, 104)`

where `{102}` denotes `$E_1$ .truelist`
containing only 102

27

Backpatching

The six statements generated so far are thus

```
100:  if a < b goto _  
101:  goto _  
102:  if c < d goto _  
103:  goto _  
104:  if e < f goto _  
105:  goto _
```

28

Backpatching

The semantic action calls

backpatch({102},104)

which fills in **104** in statement **102**.

29

Backpatching

The call fills in 104 in statement 102

```
100: if a < b goto _  
101: goto _  
▶ 102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```

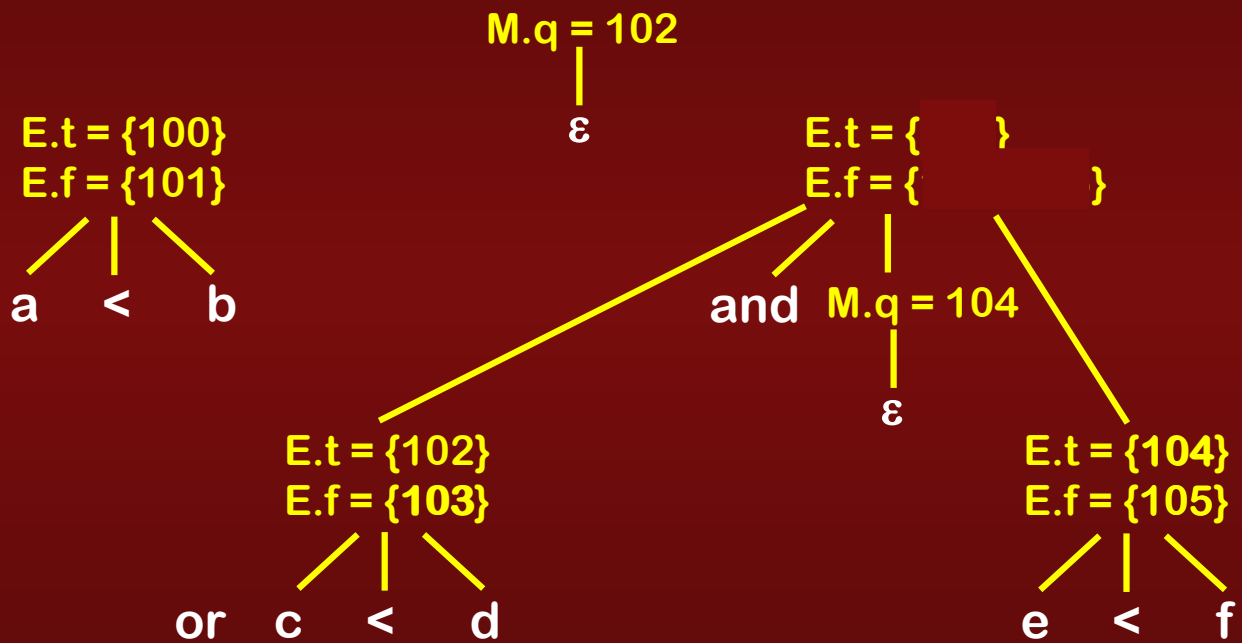
30

Semantic Actions

$E \rightarrow E_1 \text{ and } M \ E_2$

```
{ backpatch(E1.truelist, M.quad);  
  E.truelist = E2.truelist;  
  E.falselist =  
    merge(E1.falselist, E2.falselist);}
```

31



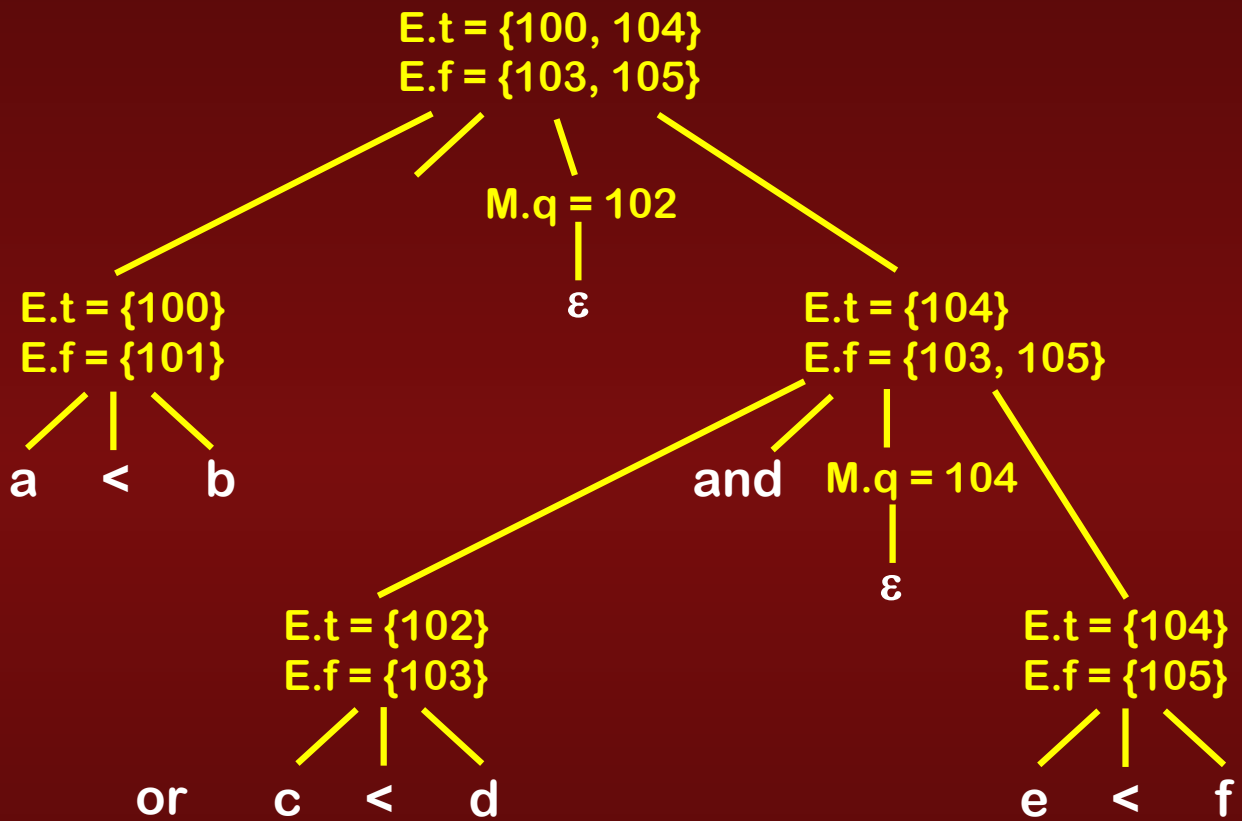
32

Backpatching

- We now reduce by the production

$$E \rightarrow E_1 \text{ or } M \ E_2$$

33



34

Semantic Actions

$E \rightarrow E_1 \text{ or } M E_2$

{

backpatch(E_1 .falselist, M.quad);

E .truelist =

merge(E_1 .truelist, E_2 .truelist);

E .falselist = E_2 .falselist;

}

35

Backpatching

The semantic action calls

backpatch({101},102)

which fills in **102** in statement **101**.

36

Backpatching

```
100:  if a < b goto _  
► 101:  goto 102  
102:  if c < d goto 104  
  
103:  goto _  
104:  if e < f goto _  
105:  goto _
```

37

Backpatching

```
100: if a < b goto _  
101: goto 102  
102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```



These instructions will have their targets filled later in the compilation

38

Semantic Actions

$E \rightarrow E_1 \text{ or } M \ E_2$

{

backpatch(E_1 .falseList, M.quad);

E .trueList =

merge(E_1 .trueList, E_2 .trueList);

E .falseList = E_2 .falseList;

}

39

Flow-of-Control Statements

- We now use **backpatching** to translate **flow-of-control** statements in **one pass**
- We will use the same list-handling procedures as before

42

Flow-of-Control Statements

$S \rightarrow$ if **E** then **S**
| if **E** then **S** else **S**
| while **E** do **S**
| begin **L** end
| **A**

$L \rightarrow$ **L** ; **S**
| **S**

43

Compiler Construction

Sohail Aslam

Lecture 41

Flow-of-Control Statements

$S \rightarrow \text{if } E \text{ then } S$

| $\text{if } E \text{ then } S \text{ else } S$

| $\text{while } E \text{ do } S$

| $\text{begin } L \text{ end}$

| A

$L \rightarrow L ; S$

| S

S denotes a statement

L is a statementlist

A is assignment

Semantic Actions

$S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

{
 ◆
 backpatch(E.truelist, M_1 .quad);
 backpatch(E.falselist, M_2 .quad);
 S.nextlist = merge(S_1 .nextlist,
 merge(**N.nextlist**, S_2 .nextlist));
}

N is marker non-terminal to introduce jump over code for **S₂**

3

Semantic Actions

$N \rightarrow \varepsilon$

{
 N.nextlist = makelist(nextQuad());
 emit('goto_');
}

The attribute **N.nextlist** records the quad number of the goto it generates

4

Semantic Actions

$M \rightarrow \varepsilon$

```
{  
    M.quad = nextQuad();  
}
```

5

Semantic Actions

$S \rightarrow \text{if } E \text{ then } M S_1$

```
{  
     backpatch(E.truelist, M.quad);  
    S.nextlist = merge(  
        E.falselist, S1.nextlist);  
}
```

If **E** is true, jump to **M.quad** which is start of code for **S₁**

6

Semantic Actions

$S \rightarrow \text{while } M_1 \text{ E do } M_2 S_1$

{

backpatch(S_1 .nextlist, M_1 .quad);
backpatch(E .truelist, M_2 .quad);
 S .nextlist = E .falselist;
emit('goto' M_1 .quad);

}

7

Semantic Actions

$S \rightarrow \text{begin } L \text{ end}$

{

S .nextlist = L .nextlist;

}

8

Semantic Actions

$S \rightarrow A$

```
{  
    S.nextlist = nil;  
}
```

initializes **S.nextlist** to an empty list

9

Semantic Actions

$L \rightarrow L_1 ; M S$

```
{  
     backpatch( $L_1$ .nextlist, M.quad);  
    L.nextlist = S.nextlist;  
}
```

Statement following **L₁** in order of execution is beginning of **S**

10

Semantic Actions

$L \rightarrow S$

```
{  
    L.nextlist = S.nextlist;  
}
```

11

Example

if $a < b$ or $c < d$ and $e < f$ then $x = y + z$ else $x = y - z$

12

if E_1 or M E_2 then $x=y+z$ else $x=y-z$

if E then $x=y+z$ else $x=y-z$

13

→	100	if a < b goto _
	101	goto 102
	102	if c < d goto 104
→	103	goto _
→	104	if e < f goto _
→	105	goto _
	106	
	107	
	108	
	109	

14

if E then M_1 $x=y+z$ else $x=y-z$ $M_1 \rightarrow \varepsilon$

{ M_1 .quad = 106 }

if E then M_1 A else $x=y-z$
{ emit('x=y+z') }

$A \rightarrow x=y+z$

if E then M_1 S_1 else A
{ S_1 .nextlist = nil }

$S_1 \rightarrow A$

if E then M_1 S_1 N else $x=y-z$
{ N.nextlist = [107]
emit('goto _') }

$N \rightarrow \varepsilon$

15

	100	if a < b goto _
	101	goto 102
	102	if c < d goto 104
	103	goto _
	104	if e < f goto _
	105	goto _
→	106	x=y+z
→	107	goto _
	108	
	109	

16

100	if a < b goto 106
101	goto 102
102	if c < d goto 104
103	goto 108
104	if e < f goto 106
105	goto 108
106	x=y+z
107	goto _
108	x=y-z
109	

19

Semantic Actions in YACC

The syntax-directed translation statements can be conveniently specified in **YACC**

The **%union** will require more fields because the attributes vary

20

Semantic Actions in YACC

The actual mechanics will be covered in the **handout** for the syntax-directed translation phase of the **course project**

21

Compiler Construction

Sohail Aslam

Lecture 42

Code Generation

- The code generation problem is the **task of mapping intermediate code to machine code**.

2

Code Generation

Requirements:

- Correctness
- Efficiency

3

Issues:

- **Input language:** intermediate code (optimized or not)
- **Target:** machine code, assembly language

4

Issues:

- Memory management: mapping names to data objects in the run-time system.
- Instruction Selection
- Instruction Scheduling
- Register allocation

5

Issues:

- Memory management: mapping names to data objects in the run-time system.
- Instruction Selection
- Instruction Scheduling
- Register allocation

6

Issues:

- Memory management: mapping names to data objects in the run-time system.
- Instruction Selection
- Instruction Scheduling
- Register allocation

7

Issues:

- Memory management: mapping names to data objects in the run-time system.
- Instruction Selection
- Instruction Scheduling
- Register allocation

8

Target: Assembly Language

General Characteristics

- **Byte-addressable** with 4-byte words
- **N** general -purpose registers
- Two-address instructions:
op source, destination

9

Target: Assembly Language

Operations:

- **MOV** source, destination
- **ADD** source, destination
- **SUB** source, destination
(dest = dest – source)
- **GOTO** address
- **CJ** conditional jump

10

Addressing Modes

	MODE	FORM	ADDRESS	ADDED COST
→	absolute	M	M	1
	register	R	R	0
	indexed	c(R)	c + contents(R)	1
	indirect register	*R	contents(R)	0
	indirect indexed	*c(R)	contents(c+contents(R))	1
	literal	#c	c	1
	stack	SP	SP	0
	indexed stack	c(SP)	c + contents(SP)	1

11

Target: Assembly Language

Instruction costs:

- The **cost** corresponds to length of instruction
- `MOV R0,R1 ; R0 = c(R1)`
has cost 1
- `MOV R5,M ; M = c(R5)`
has cost 2: 1 for instruction, 1 **additional** for memory address

12

Simple Code Generation

- Define a target code sequence for each intermediate code statement type.

13

Simple Code Generation

[illegible]

Simple Code Generation

Intermediate	becomes...
$a = b + c$	MOV b,a ADD c,a

- Consider the C statement: `a[i] = b[c[j]];`

Problems

- **Local decisions** do not produce good code.
- Do not take temporary variables into account

Optimize

- Get rid of the temporaries:

```
MOV addr(c), R0
ADD j, R0
MOV addr(b), R1
ADD *R0, R1
MOV addr(a), R2
ADD i, R2
MOV *R1, *R2
```

The **cost** of this code is **12**.

18

a[i] = b[c[j]];

```
t1 := c[j]  MOV addr(c), R0
            ADD j, R0
            MOV *R0, t1
t2 := b[t1]  MOV addr(b), R0
            ADD t1, R0
            MOV *R0, t2
a[i] := t2   MOV address(a), R0
            ADD i, R0
            MOV t2, *R0
```

The **cost** of this code is **18**

```
MOV addr(c), R0
ADD j, R0
MOV addr(b), R1
ADD *R0, R1
MOV addr(a), R2
ADD i, R2
MOV *R1, *R2
```

The **cost** of this code is **12**.

19

Can optimize further ...

```
MOV addr(c), R0
ADD j, R0
MOV addr(a), R2
ADD i, R2
MOV *addr(b)(R0), *R2
```

The **cost** of this code is **10**.

20

Can optimize further ...

- What is needed is a way to **generate** machine code based on **past** and **future use** of the data.

21

Compiler Construction

Sohail Aslam

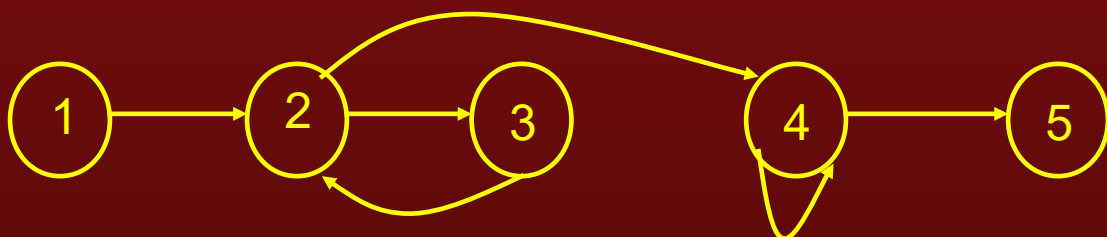
Lecture 43

Control Flow Graph - CFG

CFG = $\langle V, E, \text{Entry} \rangle$, where

V = vertices or nodes, representing an instruction or basic block (group of statements).

$E = (V \times V)$ edges, potential flow of control
Entry is an element of V , the unique program entry



Basic Blocks

A basic block is a **sequence** of **consecutive** statements with **single** entry/single exit

3

Basic Blocks

- Flow of **control** only **enters** at the beginning
- Flow of control only **leaves** at the **end**
- Variants: single entry/multiple exit, multiple entry/single exit

4

Generating CFGs

- Partition intermediate code into basic blocks
- Add control flow between blocks
 - Unconditional goto
 - Conditional goto
 - No goto at end – control passes to first statement of next block

5

Generating CFGs

- Partition intermediate code into basic blocks
- Add edges corresponding to control flow between blocks
 - Unconditional goto
 - Conditional goto – multiple edges
 - No goto at end – control passes to first statement of next block

6

Algorithm: Partition into basic blocks

Input.

sequence of three-address statements

Output.

A list of basic blocks with each three-address statements in exactly one block

7

Method.

1. Determine the set of **leaders** – the first statements of basic blocks. The rules are
 - i. The first statement is a leader
 - ii. Any statement that is the target of a conditional or unconditional goto is a leader
 - iii. Any statement that immediately follows a goto or conditional goto is a leader

8

Method.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

9

Example

Consider the C fragment for computing **dot product** $\mathbf{a}^T \mathbf{b}$ of two vectors $\mathbf{a}$ and $\mathbf{b}$ of length 20

$$\begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{20} \end{bmatrix} \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ . \\ . \\ \mathbf{b}_{20} \end{pmatrix} = \mathbf{a}_1 \mathbf{b}_1 + \mathbf{a}_2 \mathbf{b}_2 + \dots + \mathbf{a}_{20} \mathbf{b}_{20}$$

Dot Product

```
prod = 0;
i = 1;
do {
    prod = prod + a[i]*b[i];
    i = i + 1;
} while ( i <= 20 );
```

11

Dot Product

```
1  prod = 0
2  i = 1
3  t1 = 4*i      /* offset */
4  t2 = a[t1]    /* a[i] */
5  t3 = 4*i
6  t4 = b[t3]    /* b[i] */
7  t5 = t2*t4
8  t6 = prod+t5
9  prod = t6
10 t7 = i+1
11 i = t7
12 if i <= 20 goto (3)
```

12

Dot Product

1	prod = 0	leader
2	i = 1	
3	t1 = 4*i	leader
4	t2 = a[t1]	ifset */
5	t3 = 4*i	/* a[i] */
6	t4 = b[t3]	/* b[i] */
7	t5 = t2*t4	
8	t6 = prod+t5	
9	prod = t6	
10	t7 = i+1	
11	i = t7	
12	if i <= 20 goto (3)	

13

Dot Product

1	prod = 0	B_1
2	i = 1	
3	t1 = 4*i	B_2
4	t2 = a[t1]	
5	t3 = 4*i	
6	t4 = b[t3]	
7	t5 = t2*t4	
8	t6 = prod+t5	
9	prod = t6	
10	t7 = i+1	
11	i = t7	
12	if i <= 20 goto (3)	

14

```
prod = 0  
i = 1
```

B₁

```
t1 = 4*i  
t2 = a[t1]  
t3 = 4*i  
t4 = b[t3]  
t5 = t2*t4  
t6 = prod+t5  
prod = t6  
t7 = i+1  
i = t7  
if i <= 20 goto B2
```

B₂

15

Quick Sort

```
void quicksort(int m, int n)  
{  
    int i,j,v,x;  
    if( n <= m ) return;  
    i=m-1; j=n; v=a[n];  
    while(true) {  
        do i=i+1; while( a[i] < v );  
        do j=j-1; while( a[j] > v );  
        if( i >= j ) break;  
        x=a[i]; a[i]=a[j]; a[j]=x;  
    }  
    x=a[i]; a[i]=a[n]; a[n]=x;  
    quicksort(m,j); quicksort(i+1,n);  
}
```

16

Quick Sort

```
void quicksort(int m, int n)
{
    int i,j,v,x;
    if( n <= m ) return;
    i=m-1; j=n; v=a[n];
    while(true) {
        do i=i+1; while( a[i] < v);
        do j=j-1; while( a[j] > v);
        if( i >= j ) break;
        x=a[i]; a[i]=a[j]; a[j]=x;
    }
    x=a[i]; a[i]=a[n]; a[n]=x;
    quicksort(m,j); quicksort(i+1,n);
}
```

17

(1) i := m - 1	(16) t7 := 4 * i
(2) j := n	(17) t8 := 4 * j
(3) t1 := 4 * n	(18) t9 := a[t8]
(4) v := a[t1]	(19) a[t7] := t9
(5) i := i + 1	(20) t10 := 4 * j
(6) t2 := 4 * i	(21) a[t10] := x
(7) t3 := a[t2]	(22) goto (5)
(8) if t3 < v goto (5)	(23) t11 := 4 * i
(9) j := j - 1	(24) x := a[t11]
(10) t4 := 4 * j	(25) t12 := 4 * i
(11) t5 := a[t4]	(26) t13 := 4 * n
(12) if t5 > v goto (9)	(27) t14 := a[t13]
(13) if i >= j goto (23)	(28) a[t12] := t14
(14) t6 := 4*i	(29) t15 := 4 * n
(15) x := a[t6]	(30) a[t15] := x

18


```

(1)  i := m - 1
(2)  j := n
(3)  t1 := 4 * n
(4)  v := a[t1]
(5)  i := i + 1
(6)  t2 := 4 * i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
(9)  j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]

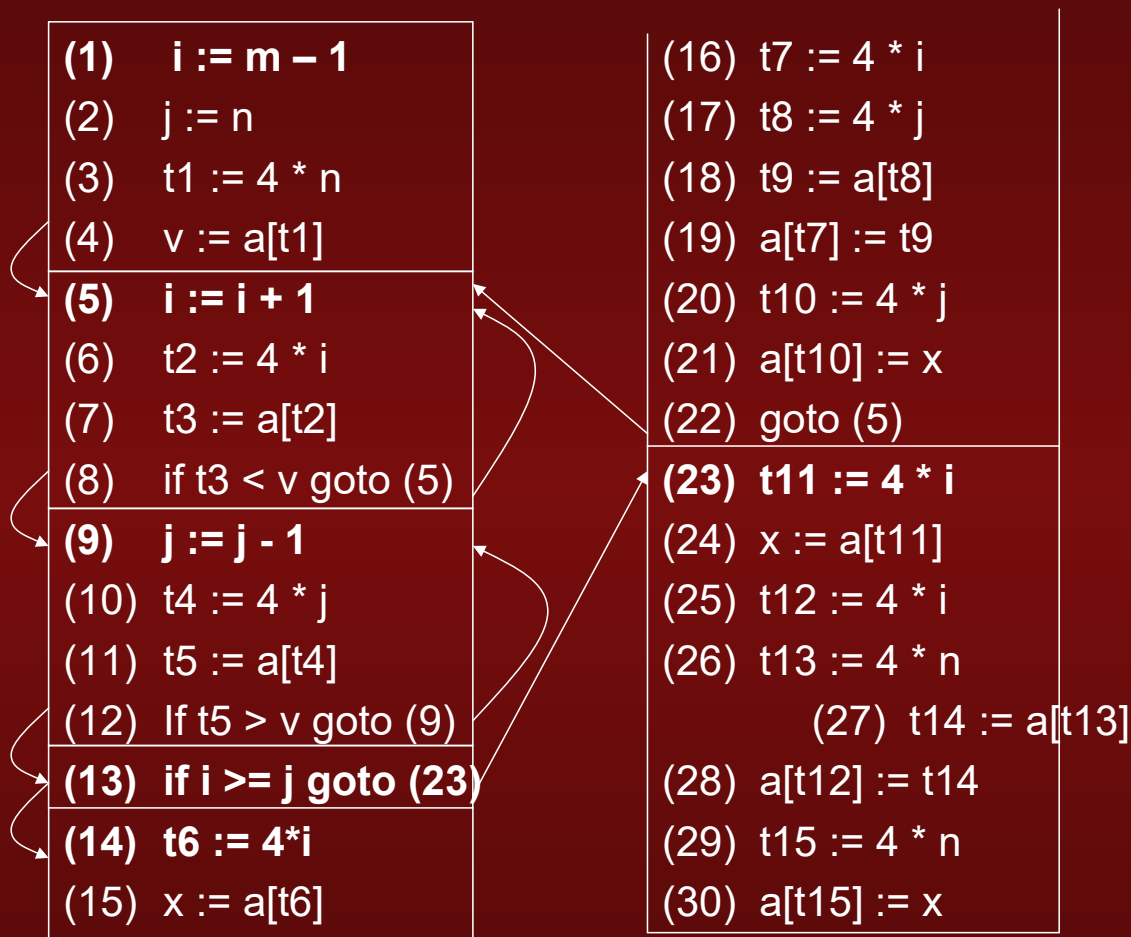
```

```

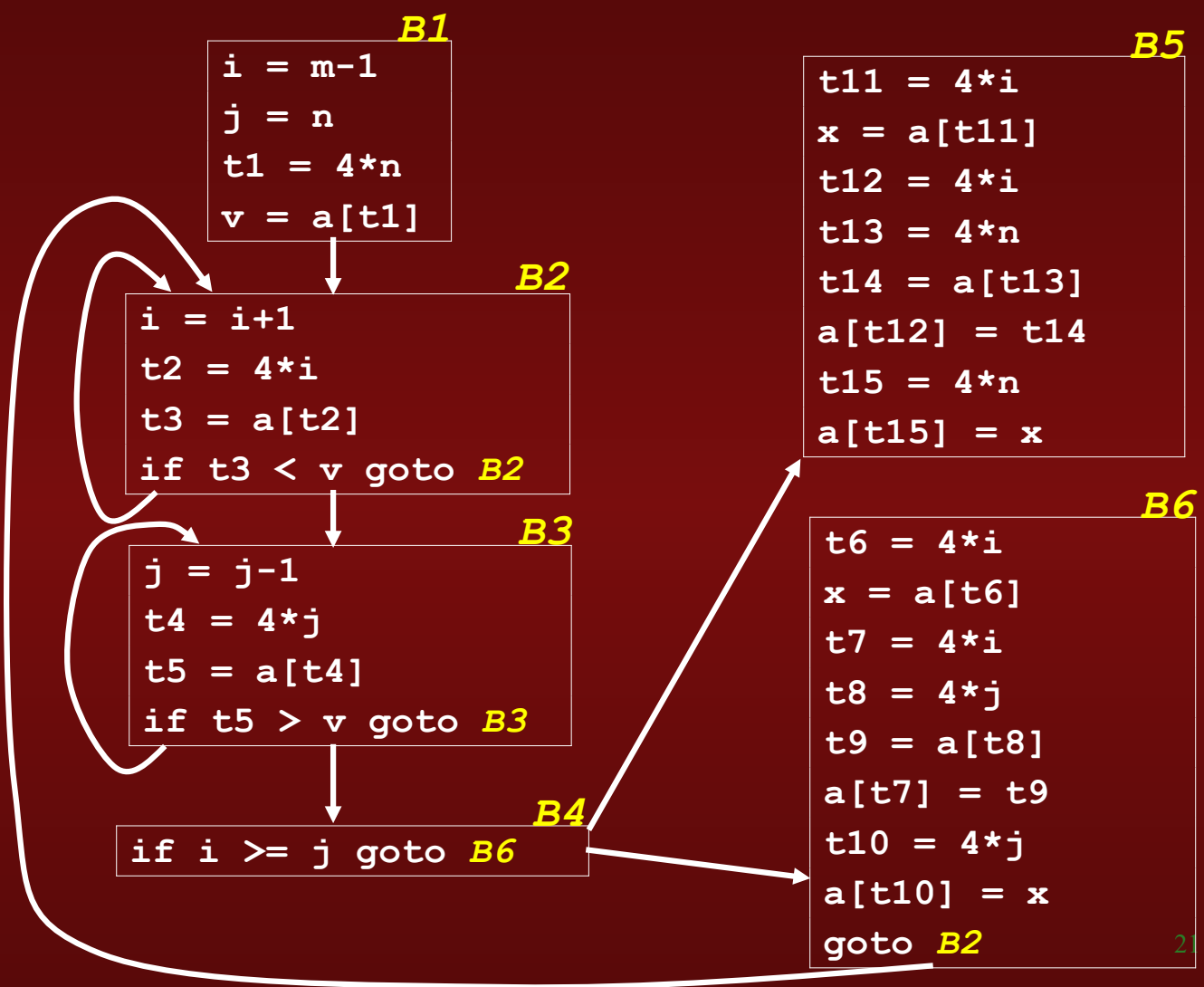
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
      (27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

```

19



20



Basic Block Code Generation

Algorithms:

- **Basic** - using liveness information
- **Using DAGS** - node numbering
- **Register Allocation**

Basic Code Generation

- Deal with **each** basic **block** individually.
- Generate code for the block using liveness information.
- Allocate registers to variables using liveness information.

23

Basic Code Generation

- Deal with **each** basic **block** individually.
- Generate code for the block using **liveness** information.
- Allocate registers to variables using liveness information.

24

Basic Code Generation

- Deal with **each** basic **block** individually.
- Generate code for the block using **liveness** information.
- At end, generate code that **saves** any **live** values left in registers.

25

Computing Live/Next Use Information

- For the statement:
$$x = y + z$$

 x has a **next use** if there is a statement s that **references** x and there is some way for **control** to **flow** from the original statement **to** s .

26

Computing Live/Next Use Information

$$\mathbf{x} = y + z$$

.....

.....

$$\mathbf{s} \quad t1 = \mathbf{x} - t3 \quad \boxed{\textit{next use}}$$

27

Computing Live/Next Use Information

- A variable is *live* at a given point in time if it **has a next use**.
- Liveness tells us whether we care about the **value held** by a variable.

28

Computing Live/Next Use Information

x = y + z *live!*

.....

.....

s t1 = x - t3

29

Computing live status

Input:

A basic block.

Output:

For each statement, set of live variables

30

Method:

- 1. Initially all **non-temporary** variables go into live **set**.
- 2. for **i** = **last** statement to **first** statement:
for statement **i**: **x = y op z**
 - attach to statement **i**, current live set.
 - remove **x** from set.
 - add **y** and **z** to set.

31

Compiler Construction

Sohail Aslam

Lecture 44

Example

$a = b + c$

$t1 = a * a$

$b = t1 + a$

$c = t1 * b$

$t2 = c + b$

$a = t2 + t2$

$a = b + c$

$t1 = a * a$

$b = t1 + a$

$c = t1 * b$

$t2 = c + b$

$a = t2 + t2$

Basic Code Generation

- Idea: deal with the instructions from **beginning** to **end**.

- Use registers whenever possible.

be discarded, freeing that register.

4

Basic Code Generation

- Idea: deal with the instructions from **beginning** to **end**.
- For **each** instruction,
 - Use **registers** whenever possible.
 - A **non-live** value in a register can be **discarded**, freeing that register.

5

Basic Code Generation

Data Structures:

- **Register descriptor** - register status (empty, inuse) and contents (one or more "values")
- **Address descriptor** - the location (or locations) where the current value for a variable can be found (register, stack, memory)

6

Basic Code Generation

Data Structures:

- **Register descriptor** - register status (empty, inuse) and contents (one or more "values")
- **Address descriptor** - the location (or locations) where the current value for a variable can be found (register, stack, memory)

7

Instruction type: $x = y \text{ op } z$

1. If y is **non-live** and in register R (alone) then generate

OP z' , R

(z' = best location for z)

8

Instruction type: $x = y \text{ op } z$

2. If operation is **commutative**, z is **non-live** and is in register R (alone), generate

OP y' , R

(y' = best location for y)

9

Instruction type: $x = y \text{ op } z$

3. If there is a free register R ,
generate

MOV y' , R
OP z' , R

10

Instruction type: $x := y \text{ op } z$

4. Use a memory location. Generate

MOV y' , x
OP z' , x

11

Instruction type: $x = y \text{ op } z$

- **Update** information about the current **best** location of x
- If x is in a **register**, update that register's information (descriptor)

12

Instruction type: $x = y \text{ op } z$

- If y and/or z are **not live** after **this** instruction, **update** register and address descriptors according.

13

Returning to live Example

```
live = {b,c}
a = b + c
live = {a}
t1 = a * a
live = {a,t1}
b = t1 + a
live = { b,t1}
c = t1 * b
live = {b,c}
t2 = c + b
live = {b,c,t2}
a = t2 + t2
live = {a,b,c}
```

14

Example

Initially

- three registers:
(-, -, -) all empty
- current values:
(a,b,c,t1,t2) = (m,m,m, -, -)

15

1: $a = b + c$, Live = a

getreg(): L = R1

MOV b, R1

ADD c, R1 ; $R1 := R1 + c$

Registers: (a, -, -)

current values: (R1, m, m, -, -)

16

2: $t1 = a * a$, Live = a, t1

L = R2 (since a is live)

MOV R1, R2

MUL R2, R2 ; $R2 = R2 * R2$

Registers: (a, t1, -)

current values: (R1, m, m, R2, -)

17

3: $b = t1 + a$, Live = $b, t1$

Since a is not live $L = R1$

ADD R2, R1 ; $R1 = R1 + R2$

Registers: ($b, t1, -$)

current values: ($m, R1, m, R2, -$)

18

4: $c = t1 * b$, Live = b, c

Since $t1$ is not live $L = R2$

MUL R1, R2 ; $R2 = R1 * R2$

Registers: ($b, c, -$)

current values: ($m, R1, R2, -, -$)

19

5: $t2 = c + b$, Live = $b, c, t2$

L = R3

MOV R2, R3

ADD R1, R3 ; $R3 = R1 + R2$

Registers: $(b, c, t2)$

current values: $(m, R1, R2, -, R3)$

20

6: $a = t2 + t2$, Live = a, b, c

ADD R3, R3

Registers: (b, c, a)

current values: $(R3, R1, R2, -, R3)$

21

End of block

- move all live variables to memory:

```
MOV R3, a  
MOV R1, b  
MOV R2, c
```

- all registers available

22

```
; a := b + c  
    LOAD b, R1  
    ADD c, R1           ; R1 := R1 + c  
; t1 := a * a  
    MOV R1, R2  
    MUL R2, R2          ; R2 = R2 * R2  
; b := t1 + a  
    ADD R2, R1          ; R1 = R1 + R2  
; c := t1 * b  
    MUL R1, R2          ; R2 = R1 * R2  
; t2 := c + b  
    MOV R2, R3  
    ADD R1, R3          ; R3 = R1 + R2  
; a := t2 + t2  
    ADD R3, R3  
    MOV R3, a           ; mov live  
    MOV R1, b           ; var to memory  
    MOV R2, c
```

23

Compiler Construction

Sohail Aslam

Lecture 45

Basic Code Generation

Liveness information allows us to **keep** values in **registers** if they will be used **later** (efficiency)

Basic Code Generation

- Why do we assume all variables are live at the end of blocks?
- Can we do better?

3

Basic Code Generation

- Why do we need to save live variables at the end?
- We might have to reload them in the next block.
- Global Data-Flow Analysis

4

DAG Representations

A *dag* (directed acyclic graph) for a **basic block** has the following labels for the nodes:

5

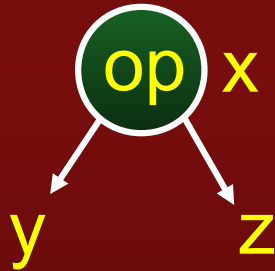
DAG Representations

- **Leaves** are labeled by unique identifiers.
- **Interior** nodes are labeled by **operator** symbols.
- Nodes can have **multiple labels** since they represent computed values.

6

DAG Representations

BB DAG for $x = y \text{ op } z$



7

Generating DAGs

For statement $i: x = y \text{ op } z$

- if $y \text{ op } z$ node exists,
add x to the label for that node.
else
add node for $y \text{ op } z$.
- If y or z exist in the dag,
point to existing locations
else
add leaves for y and/or z and have
the op node point to them.

8

Generating DAGs

- Label the **op** node with **x**.
- If **x** existed previously as a leaf, **subscript** that previous entry.
- If **x** is associated with other interior nodes, remove them from that list.

9

Example 1

$a = b + c$

$t1 = a * a$

$b = t1 + a$

$c = t1 * b$

$t2 = c + b$

$a = t2 + t2$

10

Example 1

$$a = b + c$$

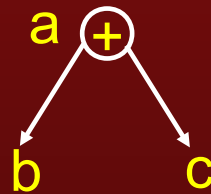
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



11

Example 1

$$a = b + c$$

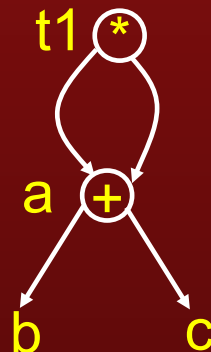
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



12

Example 1

$$a = b + c$$

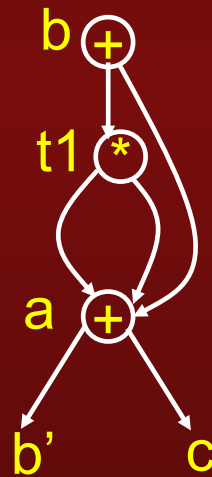
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



13

Example 1

$$a = b + c$$

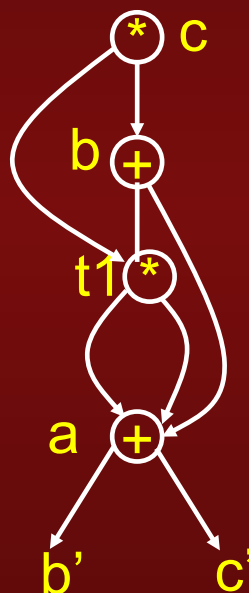
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



14

Example 1

$$a = b + c$$

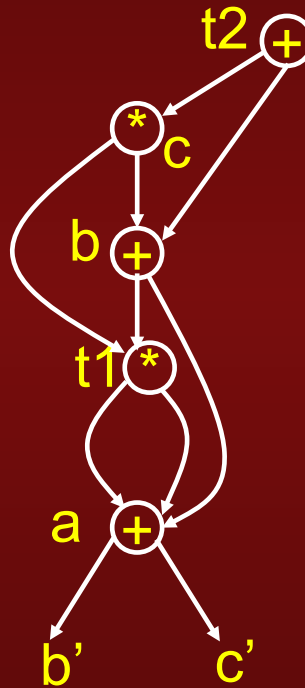
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



15

Example 1

$$a = b + c$$

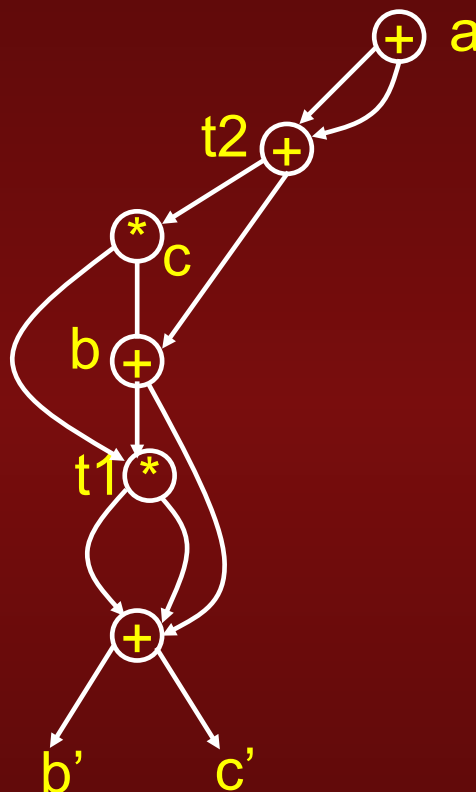
$$t1 = a * a$$

$$b = t1 + a$$

$$c = t1 * b$$

$$t2 = c + b$$

$$a = t2 + t2$$



16

Example 2

$t1 = 4 * i$

$t2 = a[t1]$

$t3 = 4 * i$

$t4 = b[t3]$

$t5 = t2 * t4$

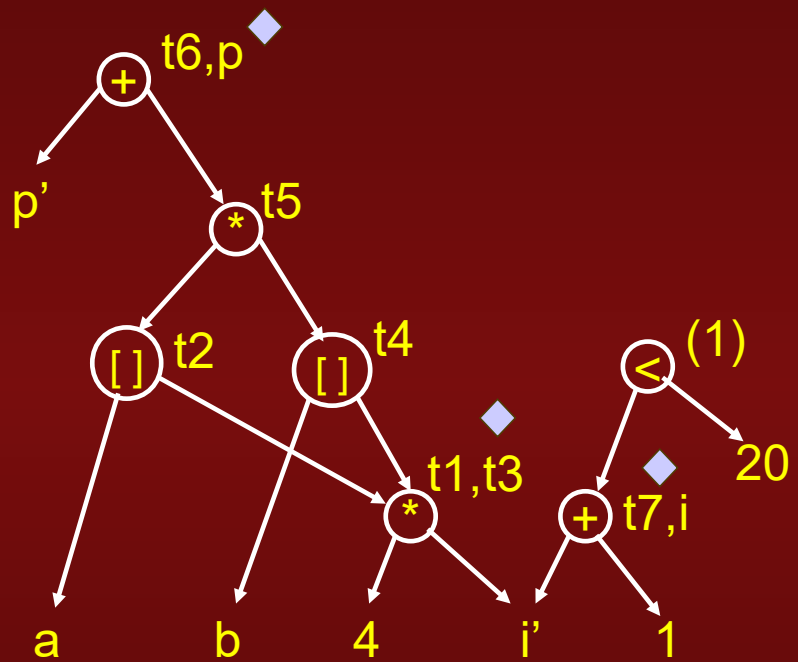
$t6 = p + t5$

$p = t6$

$t7 = i + 1$

$i = t7$

if $i < 20$ goto (1)



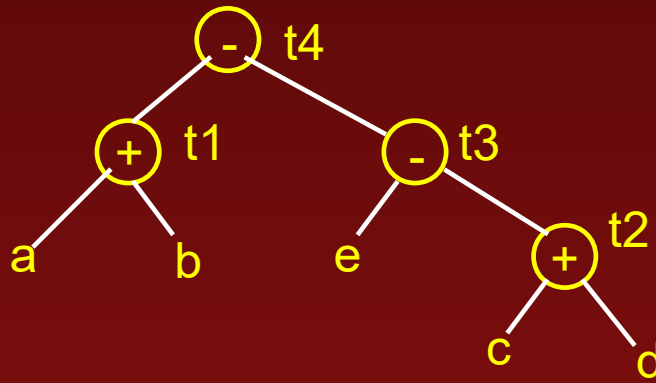
17

DAGs and optimization

- Detect **common sub-expression** elimination. Node with more than one parent is common sub-expression
- **Reordering** to reduce number of generated instructions
 - node listing
 - labeling algorithms

18

Order Matters



Order 1

$t1 = a + b$
 $t2 = c + d$
 $t3 = e - t2$
 $t4 = t1 - t3$

Order 2

$t2 = c + d$
 $t3 = e - t2$
 $t1 = a + b$
 $t4 = t1 - t3$

19

Order #1 with 2 registers

$t1 = a + b$
 $t2 = c + d$
 $t3 = e - t2$
 $t4 = t1 - t3$

MOV a, R0
ADD b, R0
MOV c, R1
ADD D, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
; 10 instructions

20

Order #2 with 2 registers

$t2 = c + d$
 $t3 = e - t2$
 $t1 = a + b$
 $t4 = t1 - t3$

MOV c, R0
ADD D, R0
MOV e, R1
SUB R0,R1
MOV a,R0
ADD b, R0
MOV R0,t4
; 7 instructions

21

Order 1

→ $t1 = a + b$
 $t2 = c + d$
 $t3 = e - t2$
→ $t4 = t1 - t3$

Order 2

$t2 = c + d$
 $t3 = e - t2$
 $t1 = a + b$ ←
 $t4 = t1 - t3$ ←

- Reordering **improved** code because computation of $t4$ immediately followed computation of $t1$, its left operand.
- $t1$ must be in a register and it is

22

Optimal Ordering Algorithms

- Heuristic ordering algorithm: node listing
- Ordering when **DAG** for a block is a **tree**
- Both algorithms presented in the book

23

Register Allocation

- How to best use the **bounded** number of **registers**.
- Local register allocation: **graph coloring** problem.

24

Register Allocation

Complications:

- special purpose registers
- operators requiring multiple registers.

25

Algorithm: K registers

1. Compute liveness information.
2. Create **interference graph G**
 - one **node** for each **variable**,
 - an **edge** connects **two variables** if **one is live** at a point where the other is defined

26

Example

live

$a = b + c \quad \{a\}$

$t1 = a * a \quad \{t1, a\}$

$b = t1 + a \quad \{b, t1\}$

$c = t1 * b \quad \{b, c\}$

$t2 = c + b \quad \{b, c, t2\}$

$a = t2 + t2 \quad \{a, b, c\}$

inteferece graph

27

Example

live

$a = b + c \quad \{a\}$

$t1 = a * a \quad \{t1, a\}$

$b = t1 + a \quad \{b, t1\}$

$c = t1 * b \quad \{b, c\}$

$t2 = c + b \quad \{b, c, t2\}$

$a = t2 + t2 \quad \{a, b, c\}$



inteferece graph

28

Example

live

$a = b + c$ $\{a\}$

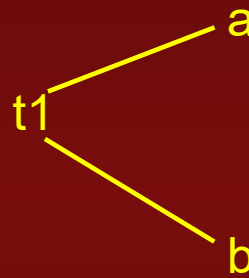
$t1 = a * a$ $\{t1, a\}$

$b = t1 + a$ $\{b, t1\}$

$c = t1 * b$ $\{b, c\}$

$t2 = c + b$ $\{b, c, t2\}$

$a = t2 + t2$ $\{a, b, c\}$



interference graph

29

Example

live

$a = b + c$ $\{a\}$

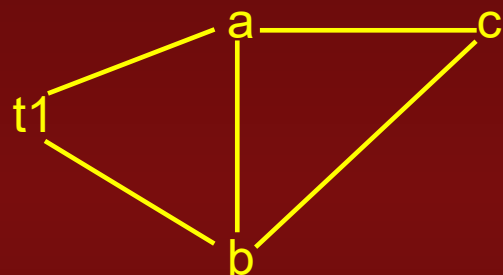
$t1 = a * a$ $\{t1, a\}$

$b = t1 + a$ $\{b, t1\}$

$c = t1 * b$ $\{b, c\}$

$t2 = c + b$ $\{b, c, t2\}$

$a = t2 + t2$ $\{a, b, c\}$



interference graph

30

Example

live

$a = b + c$ {a}

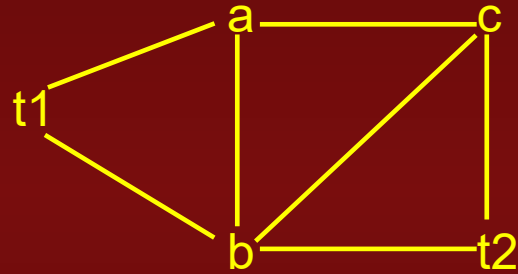
$t1 = a * a$ {t1,a}

$b = t1 + a$ {b,t1}

$c = t1 * b$ {b,c}

$t2 = c + b$ {b,c,t2}

$a = t2 + t2$ {a,b,c}



interference graph

31

Example

live

$a = b + c$ {a}

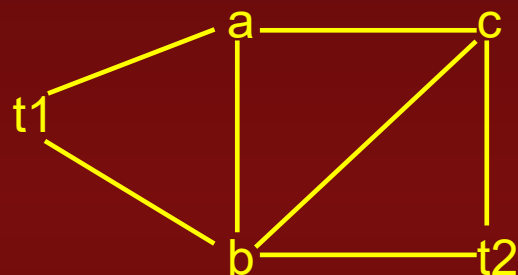
$t1 = a * a$ {t1,a}

$b = t1 + a$ {b,t1}

$c = t1 * b$ {b,c}

$t2 = c + b$ {b,c,t2}

$a = t2 + t2$ {a,b,c}



interference graph

32

3. Simplify - For any **node m** with less than **K** neighbors, remove it from the graph and push it onto a **stack**. If **$G - m$** can be colored with **K** colors, so can **G** . If we reduce the entire graph, goto step 5.

33

4. **Spill** - If we get to the point where we are left with only nodes with degree **$\geq K$** , mark some node for potential spilling (to memory). Remove and push onto stack. Back to step 3.

34

5. **Assign colors** - Starting with empty graph, **rebuild graph** by **popping** elements off the **stack** and assigning a color different from neighbors.

35

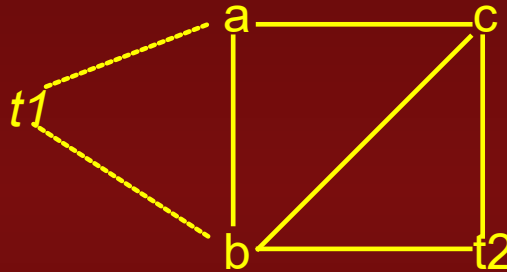
5. (**continued**)
Potential spill nodes may or may not be colorable.

Process may require iterations and rewriting of some of the code to create more temporaries.

36

Example, $k = 3$

Assume $k = 3$

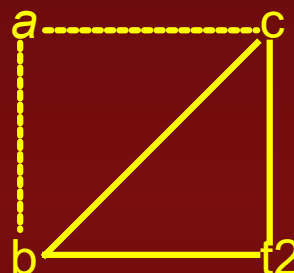


Remove t_1

37

Example

Assume $k = 3$



Remove a

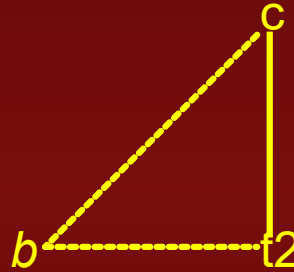
38

Example

Assume $k = 3$



Remove b



39

Example

Assume $k = 3$



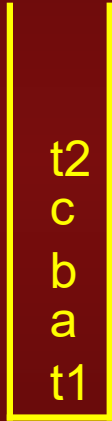
Remove c



40

Example

Assume $k = 3$



Remove t2

41

Rebuild Graph

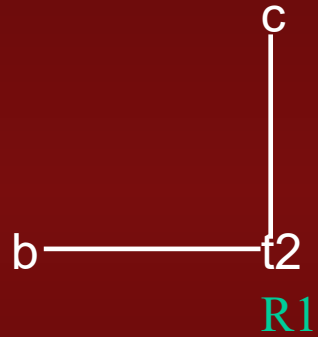
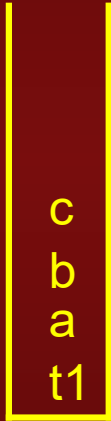
Assume $k = 3$



42

Example

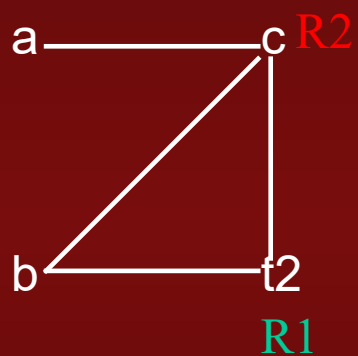
Assume $k = 3$



43

Example

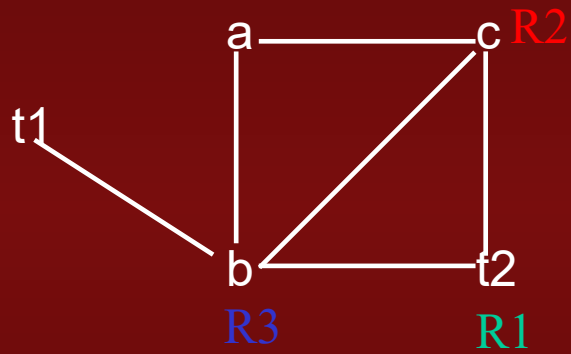
Assume $k = 3$



44

Example

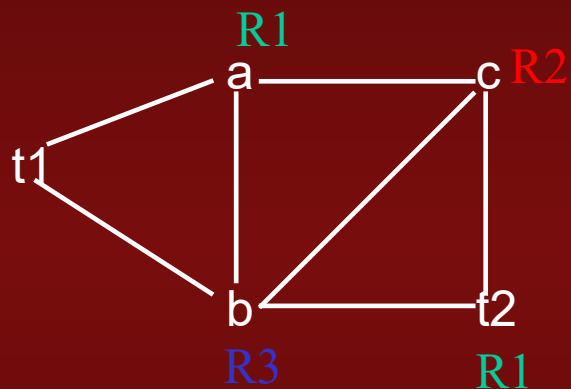
Assume $k = 3$



45

Example

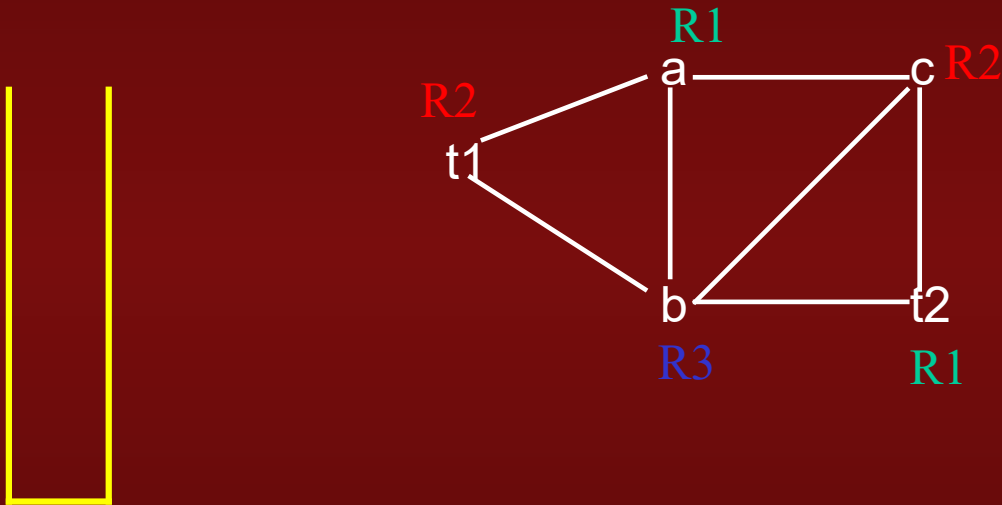
Assume $k = 3$



46

Example

Assume $k = 3$



chromatic number is 3

47

Compiler Construction

End

www.vuCtn.com

Good Luck

Regards : Zarva Chaudhary

Admin:

* Zarva Chaudhary *

* Chaudhary Moazzam *

* Laiba Maki *