Systems Software

Charles E. Hughes

COP 3402 – Spring 2011

Notes

OVERVIEW
(DAYS #1,2)

Who, What, Where and When

- Instructor: **Charles Hughes;**
  Harris Engineering 247C; 823-2762
  (phone is not a good way to get me);
  Office Hours: **TR 9:45AM-11:15AM**
  charles.e.hughes@knights.ucf.edu
  (e-mail is a good way to get me)
  Subject: COP3402
- GTA: **Remo Pillat; rpillat@knights.ucf.edu**
  Office Hours: **M 2:30PM-4:30PM**
- Web Page:
  http://www.cs.ucf.edu/courses/cop3402/spring2011
- Meetings: **TR 12:00PM-1:15PM, HEC-118;**
  28 periods, each 75 minutes long.
  Final Exam is separate from class meetings
- Labs: **S11:R8:30-9:20; S12:R 9:30-10:20; HEC110**
- Final exam: **Thursday, April 28, 10:00AM – 12:50PM**

Text Material

- Textbook: *System Software Knights*, University of Central Florida Custom Edition, Pearson Custom Publishing 2008, ISBN 978-0-555-04647-0. Taken from:
  - *System Software: An Introduction to Systems Programming,* Third Edition by Leland Beck.
  - *Concepts of Programming Languages,* Eighth Edition by Robert W. Sebesta.
  - *Compilers: Principles, Techniques, & Tools,* Second Edition by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
  - *Operating Systems: Internals and Design Principles,* Sixth Edition by William Stallings.
- Other material linked from web site and in these notes

Goals of Course

- **Course Outline:** This course is designed to provide a fundamental understanding of real and virtual machines as language processor. We will study the processor as an instruction interpreter. Compilers, assemblers, and virtual machines will be presented as systems software for program development. An introduction to operating system will be given. The course is a blend of theory and practice, with a heavy dose of both.

- **Course Topics:** introduction to compilers and interpreters, virtual machines, computer architecture and assembler, loaders and linkers, macro-preprocessors, run time environments and operating systems

- **Prerequisites:** COP 3502 – Computer Science I

Expected Outcomes

- You will gain a solid understanding of various types of systems software (purpose, challenges, theoretical framework, various options for implementation).
- You will have a strong sense of the computational bounds that drive various strategies and compromises.
- You will hone your skills as software designers and programmers.
- You will (hopefully) come away with stronger formal proof skills and a better appreciation of the importance of discrete mathematics to all aspects of CS.

Keeping Up

- I expect you to visit the course web site regularly (preferably daily) to see if changes have been made or material has been added.
- Attendance is preferred, although I do not typically take role. Role may be taken if attendance and interaction drops off. This is also true of the labs.
- I do ask lots of questions in class and give lots of hints about the kinds of questions I will ask on exams. It would be a shame to miss the hints, or to fail to impress me with your insightful in-class answers.
- You are responsible for all material covered in class, whether in the text or not.

Rules to Abide By

- Do Your Own Work
  - When you turn in an assignment, you are implicitly telling me that these are the fruits of your labor. Do not copy anyone else's homework or let anyone else copy yours. In contrast, working together to understand lecture material and solutions to problems not posed as assignments is encouraged.
- Late Assignments
  - I will accept late assignments up to two days past the due date, except for the final project for which no leeway will be given. Lateness has its consequences as seen on the grading policy.
- Exams
  - No communication during exams, except with me or a designated proctor, will be tolerated. A single offense will lead to termination of your participation in the class, and the assignment of a failing grade (F or possibly ZF, see http://z.ucf.edu/).

Make Ups and Extra Credits

- Exams can only be made up under extreme extenuating circumstances. Traffic and malfunctioning alarm clocks are not valid excuses. If you miss an exam, you are responsible for contacting the instructor immediately. If you have not contacted the instructor within one day of the exam, you cannot make it up even if you had a legitimate reason for missing the exam, unless the circumstances preventing you from taking the exam also caused you to be unable to contact the instructor.
- I don't do extra credits unless I do them for the whole class and that is very, very rare.

Evaluation (tentative)

- **Grading Policy:**
  - (20%) Programming and Other Assignments
  - (20%) Mid-term Exam(s)
  - (30%) Final Exam
  - (25%) Final Programming Project
  - (5%) Open as to where that will go
  - The weights of exams will be adjusted to your personal benefits, as I reward good trends and downplay (but don't totally disregard) bad anomalies.
- Each assignment will have a due date and 10% will be subtracted for each day late (up to 2 days late, 20% off; more than two days late results in no credit).
- Grading will be A >= 90%, B+ >= 87%, B >= 80%, C+ >= 77%, C >= 70%, D >= 60%, F < 60%

Important Dates

- Drop/Swap – Thursday, Jan. 13
- Exam#1 – Tuesday, Feb. 22 (tentative)
- Withdraw Deadline – Friday, March 4
- Spring Break – March 7-12
- There may be a second midterm; I will decide right after Spring Break.
- Final – Thurs, April 28, 10:00AM–12:50PM

System Software

- Systems Software consists of programs that support the operation of a computer system, help simplify the programming process and create an environment to run application software efficiently.
- Examples of systems software include:
  - Text Editors and Integrated Development Environments (IDEs)
  - Language Processors (compilers, interpreters, analyzers, …)
  - Linkers and Loaders
  - Debuggers
  - Assemblers and Just-In-Time Translators (JITs)
  - Operating Systems

Categories of Sys Software

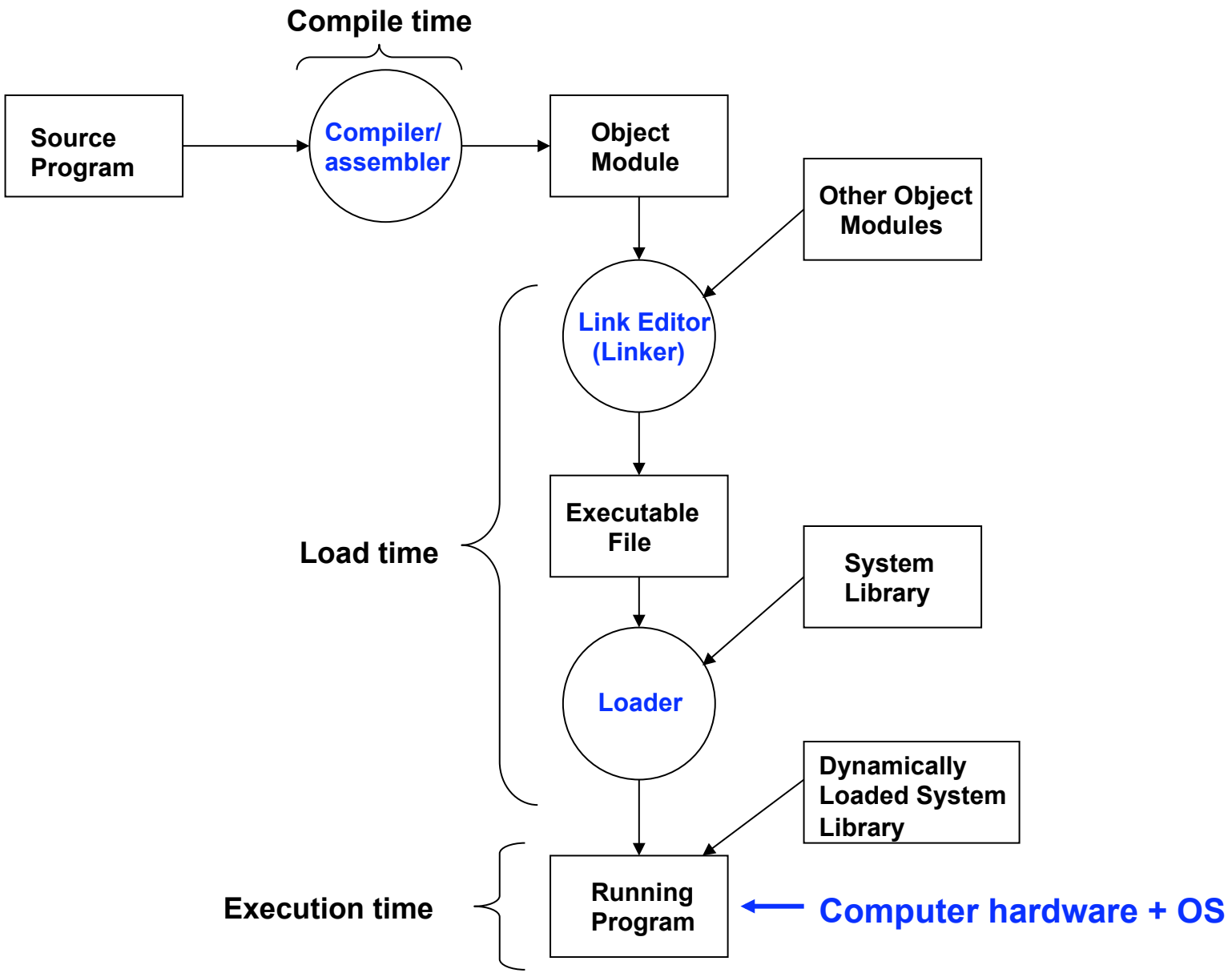
- Components for program development
  - Text Editor
  - Macro Preprocessor
  - Compiler
  - Assembler
  - Linker
  - IDE
- Components for run-time
  - Loader
  - Dynamic Linker
  - Debugger
  - Operating System

Program Development

- Text editor: Permits the creation and editing of text files (e.g. application programs).
- Macro Preprocessor: Expands macros and other directives either as part of immediate source analysis or as part of language translation.
- Compiler: Translates programs written in a high level language to object or machine code (sometimes for an abstract machine).
- Assembler: Translates programs written in assembly language to object or machine code.
- Static Linker: Combines and resolves references between object programs and creates the executable code.
- IDE: Integrates all of above in a language-aware context.

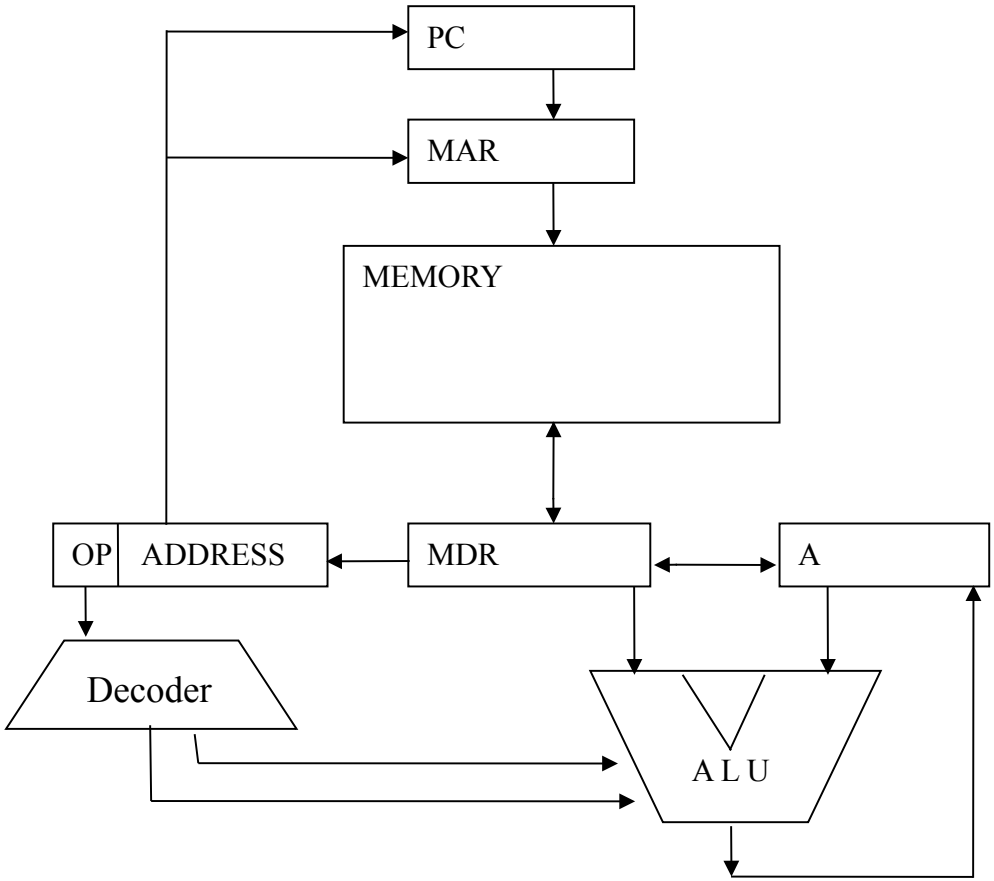
Run-time

- Loader: Loads and starts execution of machine code
- Dynamic Linker: Loads and links shared libraries at run-time.
- Debugger: Helps to debug executable programs using object code and (usually) symbolic information from source program.
- Operating System: An event driven program that makes an abstraction of the computer system. The operating system handles all resources efficiently, creates an environment in which application programs can run, and provides a friendly interface between the user and the underlying computer system.


Compile time
Source Program
Compiler/ assembler
Object Module
Other Object Modules
Link Editor (Linker)
Executable File
Load time
System Library
Loader
Dynamically Loaded System Library
Execution time
Running Program
Computer hardware + OS

A BRIEF INTRODUCTION TO MACHINE ORGANIZATION

Von Neumann Machine



Instruction Cycle

- The Instruction Cycle, or Machine Cycle, in the Von-Neumann Machine (VN) is composed of 2 steps:
  1. **Fetch Cycle:** Instruction is retrieved from memory.
  2. **Execution Cycle:** Instruction is executed.

- A simple Hardware Description Language will be used in order to understand how instructions are executed in VN.

Simple Processor Model

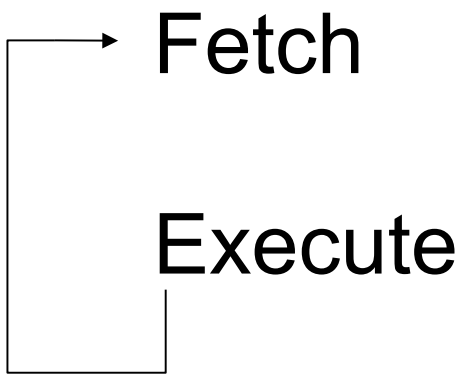
- **Program Counter (PC)** is a register that holds the address of the next instruction to be executed.
- **Instruction Register (IR)** is a register that stores the instruction to be executed by the processor.
- **DECODER** is a circuit that decides which instruction the processor will execute. For example, it takes the instruction op-code from the IR as input and outputs a signal to the ALU to control the execution of the ADD instruction.
- **Arithmetic Logic Unit (ALU)** is used to execute mathematical instructions such as ADD or SUB.
- **Accumulator (A)** is used to store data to be used as input to the ALU. (usually there are many registers for this purpose)

Simple Memory Model

- **Main Storage (MEM)** is used to store programs and data. Random Access Memory (RAM) is an implementation of MEM.
- **Memory Address Register (MAR)** is a register used to store the address to a specific memory location in Main Storage so that data can be written to or read from that location.
- **Memory Data Register (MDR)** is a register used to store data that is being sent to or received from the MEM. The data that it stores can either be in the form of instructions or simple data such as an integer.

Fetch-Execute Cycle

- In the VN, the Instruction Cycle is defined by the following loop:

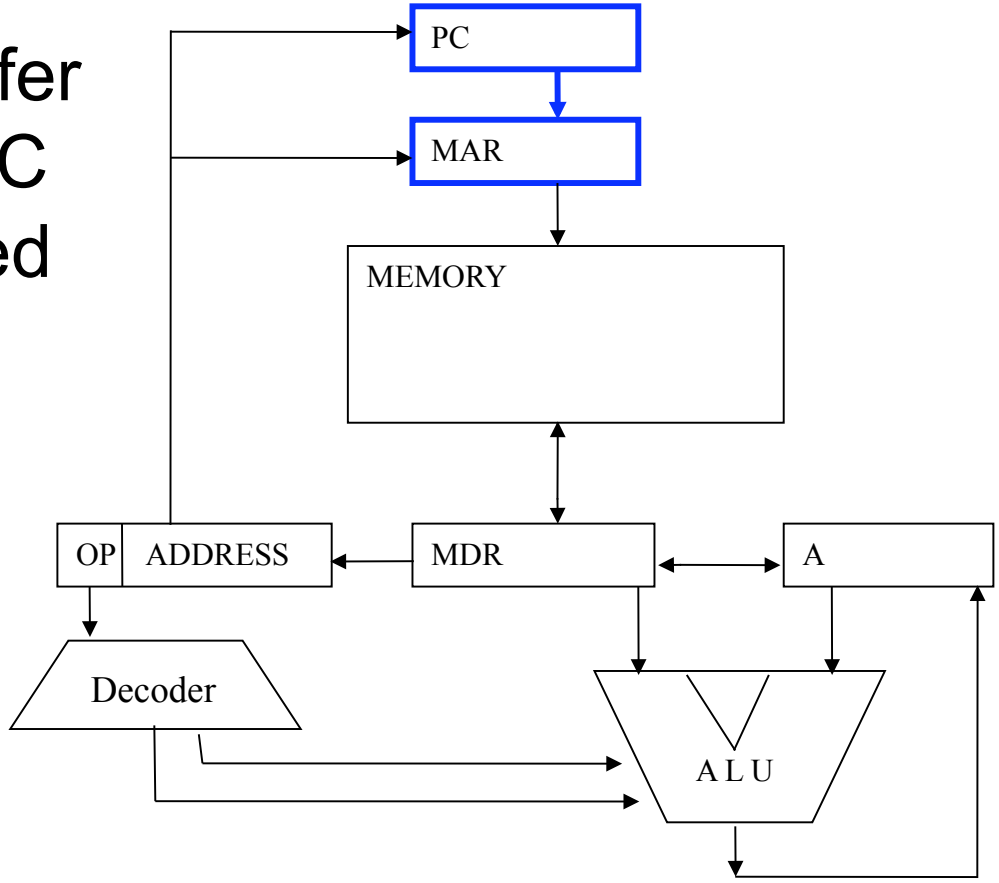


- In order to fully explain the Fetch Cycle we need to study the details of the VN data flow. The data flow consists of 4 steps.

Data Movement 1

- Given registers PC and MAR, the transfer of the contents of PC into MAR is indicated as:

  MAR←PC

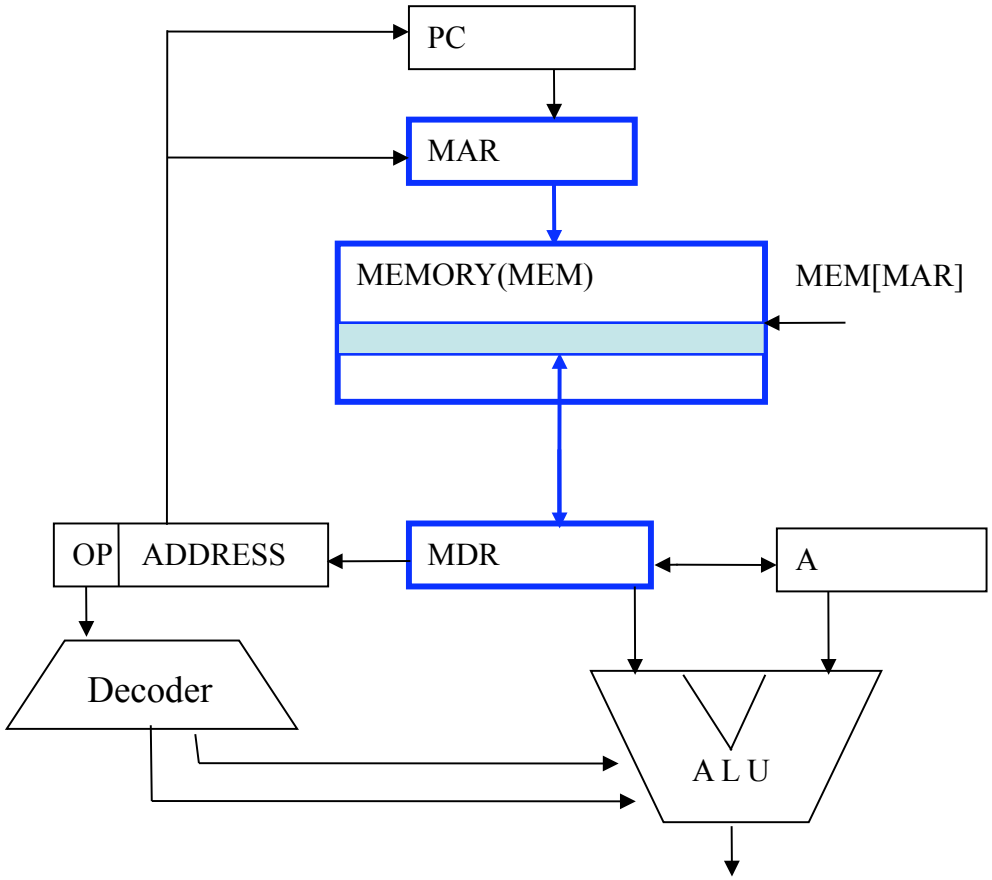


Data Movement 2

- To transfer information from a memory location to the register MDR, we use:

  MDR←MEM[MAR]

- The address of the memory location has been stored previously into the MAR register



Data Movement 2 (Cont.)

- To transfer information from the MDR register to a memory location, we use:

  MEM [MAR] ←MDR

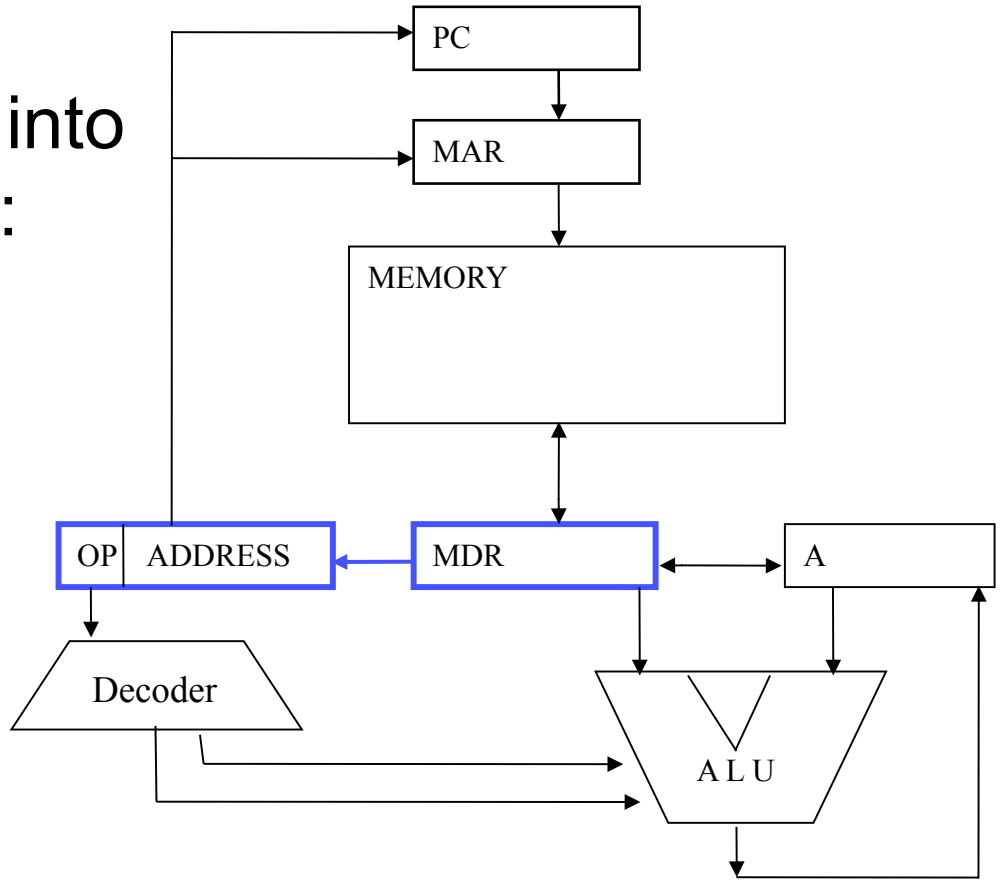
  *see previous slide for diagram

- The address of the memory location has been previously stored into the MAR

Data Movement 3

- Transferring the contents of MDR into IR is indicated as:

  IR←MDR



Instruction Register

- The Instruction Register (IR) has two fields:

  Operator (OP) and ADDRESS.

- These fields can be accessed using the selector operator “**.**”

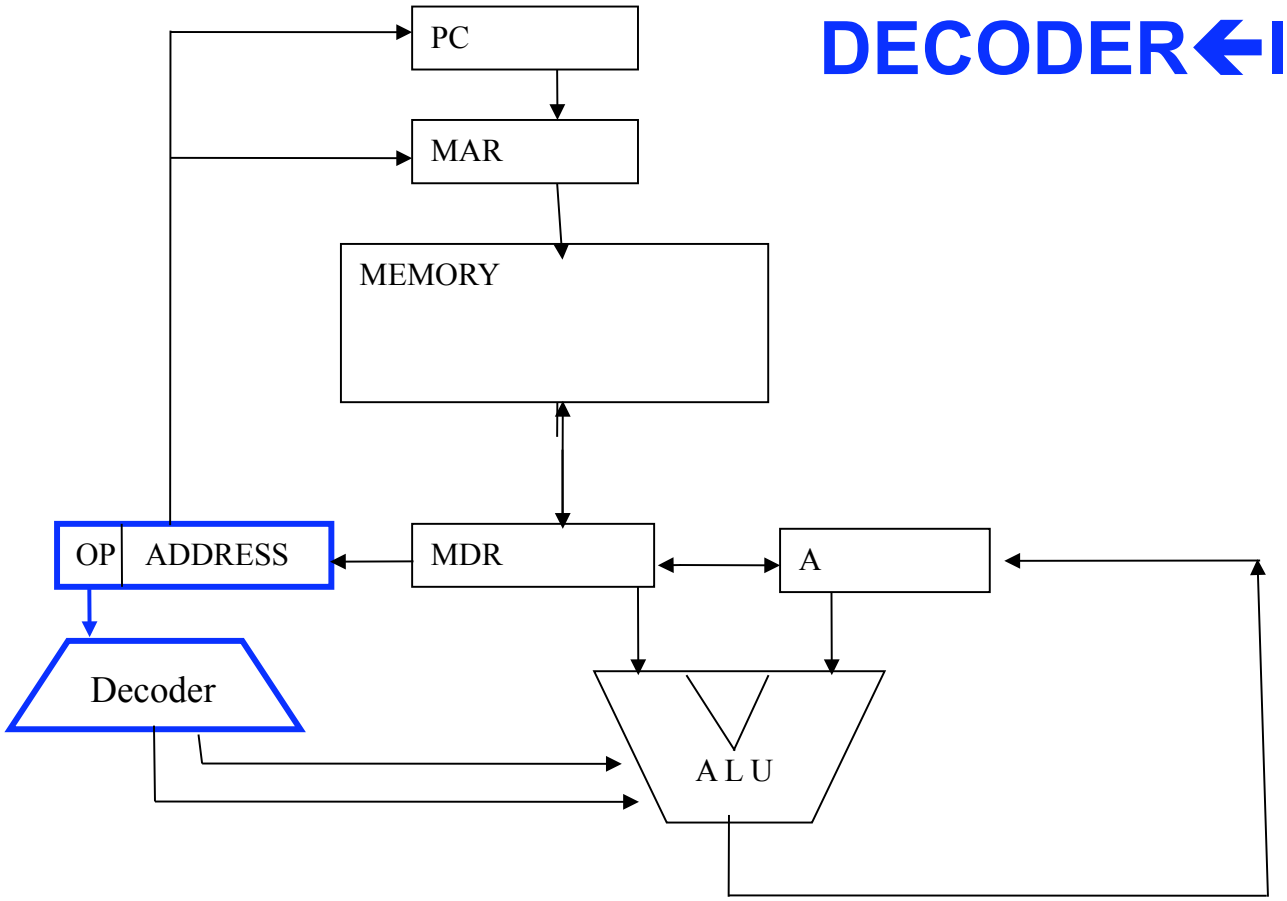
Data Movement 4

- The Operation portion of the field is accessed as IR.OP
- The operation field of the IR register is sent out to the DECODER using:

  DECODER←IR.OP

- DECODER: If the value of IR.OP==00, then the decoder can be set to execute the fetch cycle again.

Data Movement 4 (Cont.)



DECODER←IR.OP

00 Fetch Cycle

- 1.MAR ←PC
- 2.MDR ←MEM[MAR]
- 3.IR ←MDR
- 4.PC ←PC+1
- 5.DECODER ←IR.OP

1. Copy contents of PC into MAR
2. Load content of memory location into MDR
3. Copy value stored in MDR to IR
4. Increment PC Register
5. Copy the OP code into the DECODER

Execution: 01 LOAD

1. MAR ←IR.ADDR
2. MDR ←MEM[MAR]
3. A ←MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Load the content of a memory location into MDR
3. Copy content of MDR into A register
4. Set Decoder to execute Fetch Cycle

Execution: 02 ADD

1. MAR ←IR.ADDR
2. MDR ←MEM[MAR]
3. A ←A + MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Load content of memory location to MDR
3. Add contents of MDR and A register and store result into A
4. Set Decoder to execute Fetch cycle

Execution: 03 STORE

1. MAR ←IR.ADDR
2. MDR ←A
3. MEM[MAR] ←MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Copy A register contents into MDR
3. Copy content of MDR into a memory location
4. Set Decoder to execute Fetch cycle

Execution: 07 HALT

1. **STOP**

1. Program ends normally

Instruction Set Arch (ISA)

00 Fetch (hidden instruction)

MAR ←PC

MDR ←MEM[MAR]

IR ←MDR

PC ←PC+1

DECODER ←IR.OP

02 Add

MAR←IR.Address

MDR ←MEM[MAR]

A ← A + MDR

DECODER ←00

01 Load

MAR←IR.Address

MDR ←MEM[MAR]

A ← MDR

DECODER←00

03 Store

MAR←IR.Address

MDR ←A

MEM[MAR] ←MDR

DECODER ←00

07 Halt

One Address Format

OP	ADDRESS
LOAD	0000 0000 0010

1-Address ISA

01 - LOAD <X>

Loads the contents of memory location “X” into the A (A stands for Accumulator).

02 - ADD <X>

The data value stored at address “X” is added to the A and the result is stored back in the A.

03 - STORE <X>

Store the contents of the A into memory location “X”.

04 - SUB <X>

Subtracts the value located at address “X” from the A and stored the result back in the A.

1-Address ISA (Cont.)

05 - IN <Device #>

A value from the input device is transferred into A.

06 - OUT <Device #>

Print out the contents of A in the output device.

Device #	Device
5	Keyboard
7	Printer
9	Screen

07 - Halt

The machine stops execution of the program.

(Return to the OS)

08 - JMP <X>

Causes an unconditional branch to address “X”.

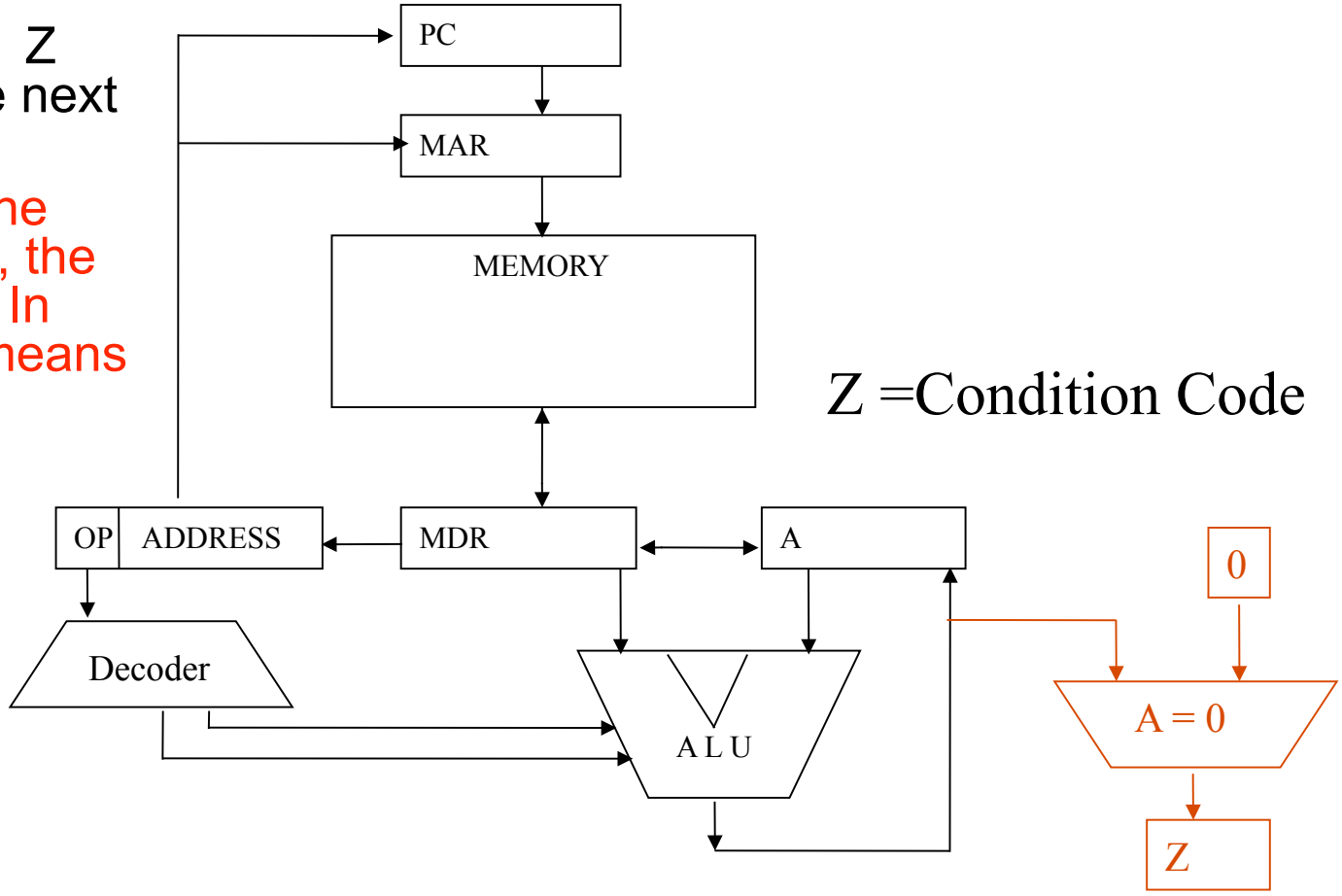
PC ← X

1-Address ISA (Cont.)

09 - SKIPZ

If the contents of Z flag = 1, skip the next instruction.

(If the output of the ALU equals zero, the Z flag is set to 1. In this machine, it means A = 0)



Condition Flags

- For this tiny assembly language, we are using only one condition code (CC) Z = 0 .
- Condition codes indicate the result of the most recent arithmetic/logical operation
- Two more flags (CC) can be incorporated to test negative and positives values:
  G = 1 Positive value
  Z = 1 Zero
  L = 1 Negative value

Program Status Word

The PSW is a register in the CPU that provides the OS with information on the status of the running program

PC	Interrupt Flags						MASK	CC			Mode
	OV	MP	PI	TI	I/O	SVC	To be defined later	G	Z	L	

In addition to the Z flag, there are two more flags:

1) G meaning “greater than zero”
2) L meaning “less than zero”

Instruction Semantics

opcode	mnemonic	meaning
0001	LOAD <x>	A ← Mem[x]
0010	ADD <x>	A ← A + Mem[x]
0011	STORE <x>	Mem[x] ← A
0100	SUB <x>	A ← A – Mem[x]
0101	IN <Device_#>	A ← read from Device
0110	OUT <Device_#>	A → output to Device
0111	HALT	Stop
1000	JMP <x>	PC ← x
1001	SKIPZ	If Z = 1 Skip next instruction
1010	SKIPG	If G = 1 Skip next instruction
1011	SKIPL	If L = 1 Skip next instruction

Sample Execution

Memory

000 Load <004>

001 Add <005>

002 Store <006>

003 Halt

004 1245

005 1755

006 0000

After execution

Memory

000 Load <000>

001 Add <001>

002 Store <002>

003 Halt

004 1245

005 1755

006 3000

1-Address Layout

- The instruction format of this one-address architecture consists of 16 bits: 4 bits to represent instructions and 12 bits for addresses :
- LOAD (opcode=0001) ADDR (here it's 17)

OP	ADDRESS
0001	**0000 0001 0001**

Assembler Language Ex.

Label	opcode	address
start	.begin	
	in	x005
	store	a
	in	x005
	store	b
	load	a
	sub	TWO
	add	b
	out	x009
	halt	
a	.data	0
b	.data	0
TWO	.data	2
	.end	start

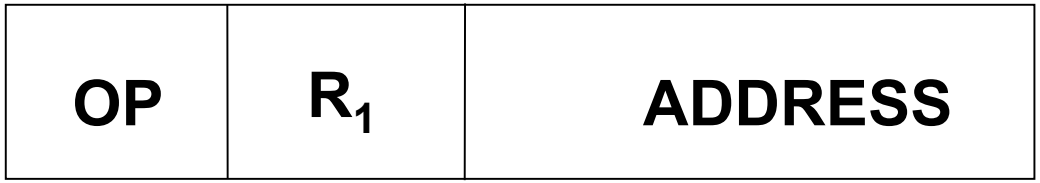
Text section (code)

Data section

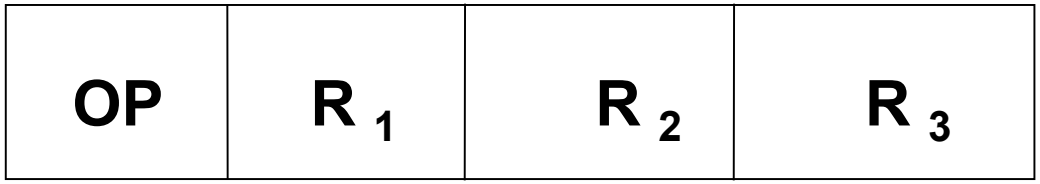
Load/Store Instr. Format

A load/store architecture has a “register file” in the CPU and might use three instruction formats. Therefore, its assembly language is different from that of the accumulator machine.

JMP <address>

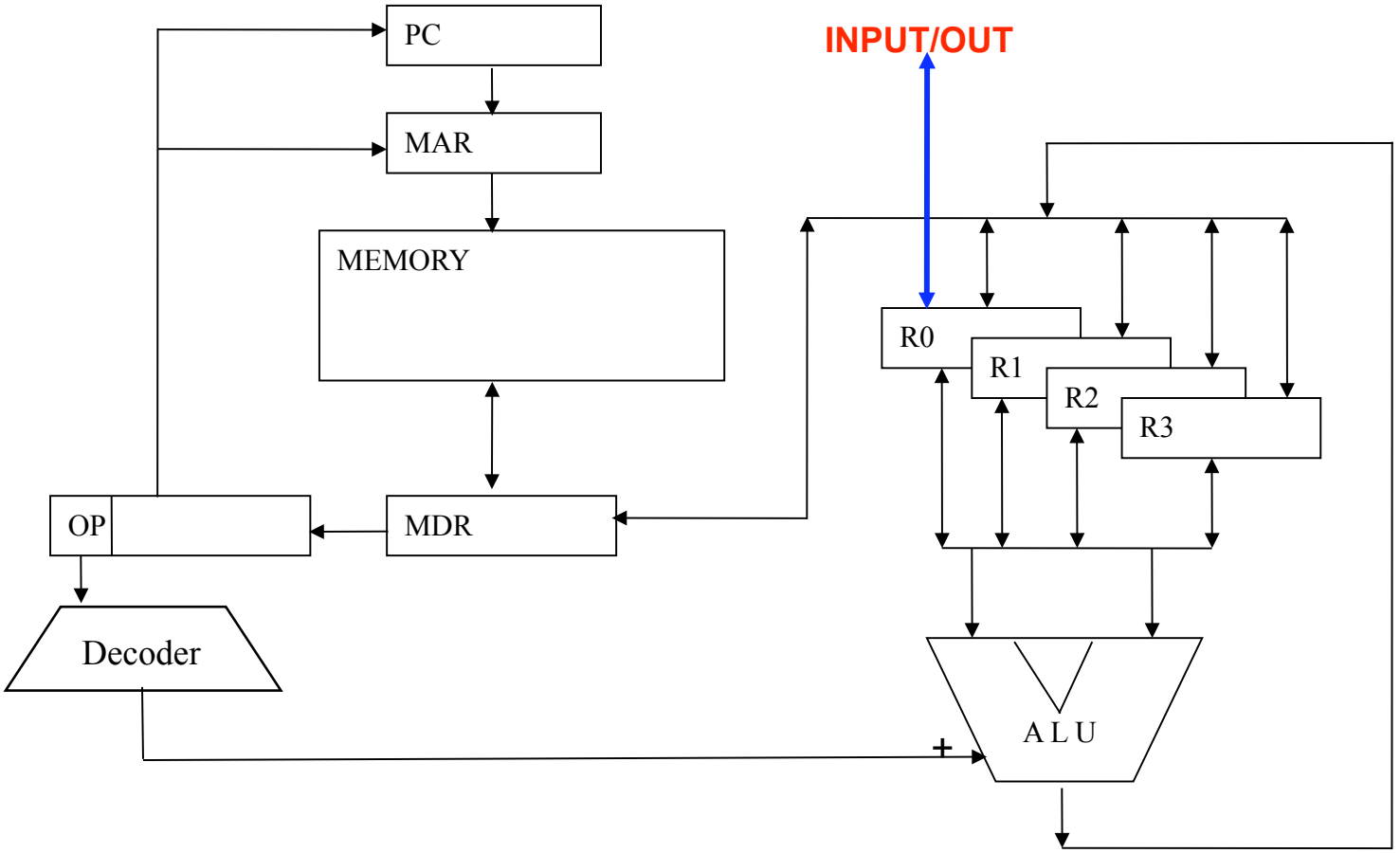


Load R1, <address>



Add R1, R2, R3

Load/Store Architecture



Comparison of Code

```
Label     opcode    address
start     .begin
          in        x005
          store     a
          in        x005
          store     b
here      load      result
          add       a
          store     result
          load      b
          sub       ONE
          store     b
          skipz
          jmp       here
          load      result
          out       x009
          halt
a         .data     0
b         .data     0
ONE       .data     1
result    .data     0
          .end      start
```

One address Architecture

```
Label     opcode    address
start     .begin
          in        x005
          store     R0, a
          in        x005
          store     R0, b
          load      R2, result
          load      R3, a
          load      R0, b
          load      R1, ONE
here      add       R2, R2, R3
          sub       R0, R0, R1
          skipz
          jmp       here
          out       R2, x009
          halt
a         .data     0
b         .data     0
ONE       .data     1
result    .data     0
          .end      start
```

Load/Store architecture

VIRTUAL MACHINE
P-CODE
(DAYS #2,3)

P Code VM

- The Pseudo-code machine is a software (virtual) machine that implements the instruction set architecture of a stack computer.
- P-code was implemented in the 70s as the target architecture for Pascal compilers. Execution was by interpretation.
- Another example of a virtual machine is the JVM (Java Virtual Machine) whose intermediate language is commonly referred to as Java bytecode.
- Another is Microsoft .NET Common Language Runtime (CLR).
- The up and comer, especially at Apple, is the Low-Level Virtual Machine (LLVM). We will look at LLVM later in course.

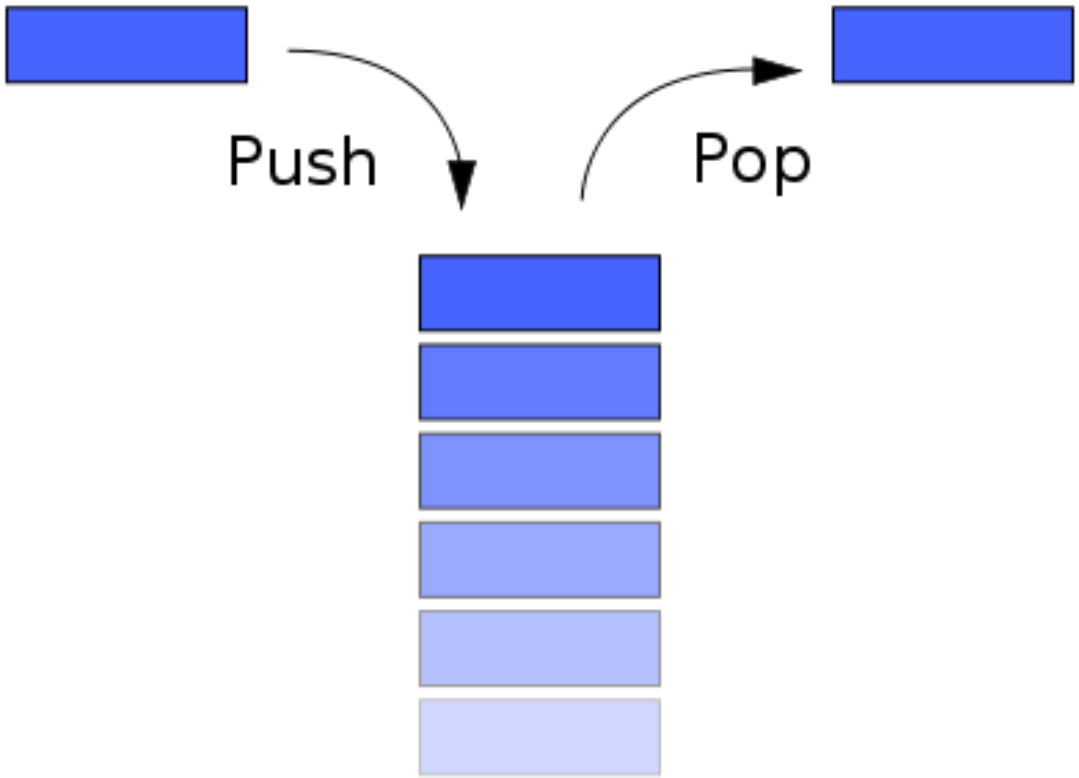
Pluses and Minuses

- Programs that have been translated to p-code are interpreted by a program that emulates the behavior of the hypothetical machine and/or compiled to machine code by a JIT (just-in-time translator).
- **Why VM ?**
  - Portability (Architecture and language independence), Simple Implementation, Compact Size, Optimizations, Debugging
- **Why not VM ?**
  - Overhead at run-time --> slower run-time (but LLVM is only 10% slower than GCC optimized code).

P-machine Architecture

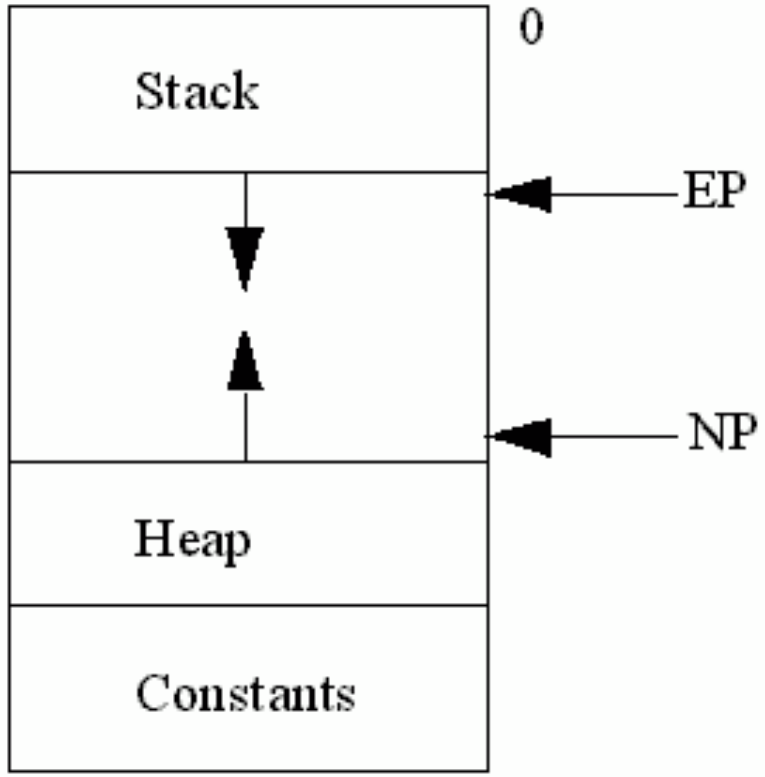
- The p-machine is a stack machine: most instructions take their operands from the stack, and place results back on the stack.
- Example: the "add" instruction replaces the two topmost elements of the stack with their sum.
- Uses one stack which is used in computation and for procedure stack frames.

Pictorial Stack Operation

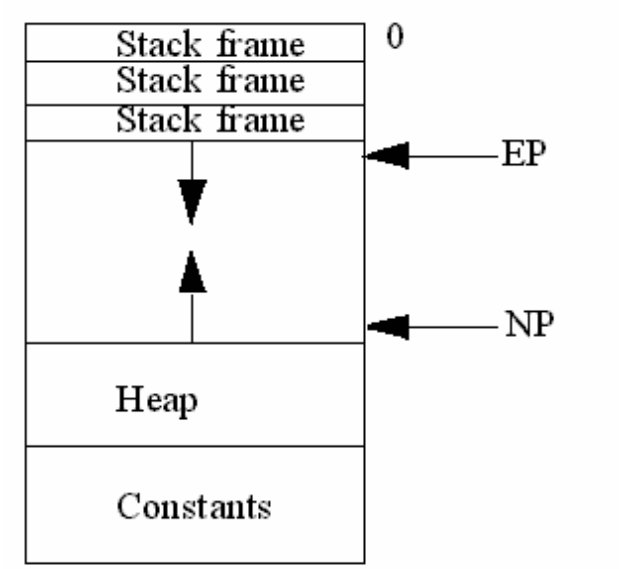


P-machine registers

- **PC** the program counter
- **SP** the stack pointer
- **MP** the mark stack pointer
- **NP** the new pointer
- **EP** the extreme stack pointer

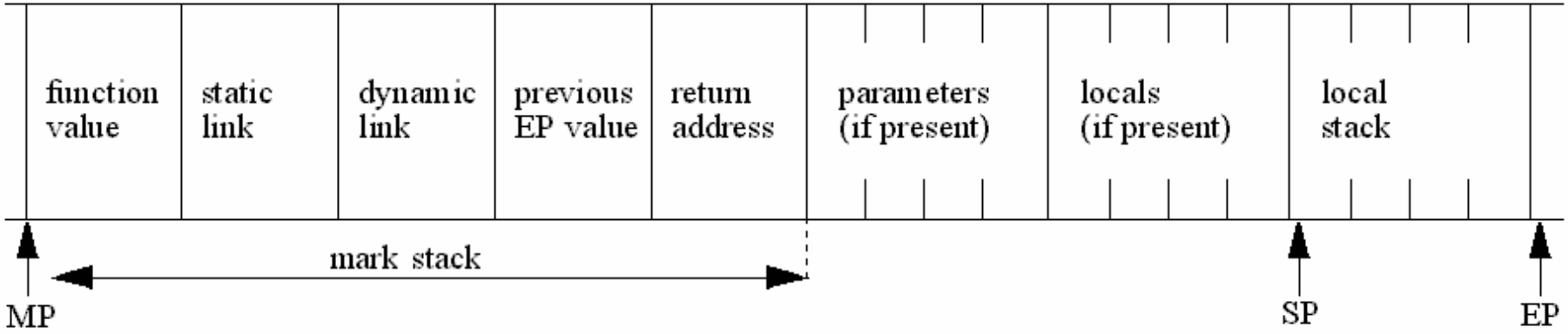


Stack Frames for P4



Static link is back to stack frame of enclosing procedure. It is needed for access to local variables of parent.

Dynamic link is back to stack frame of who called this routine.



Stack Frame (low to high)

MP -> function return value space (if needed)
static link (MP of enclosing procedure)
dynamic link (previous MP)
previous EP
return address (previous PC)
parameters (variable size)
locals
SP -> somewhere past or at end of locals
expansion space for local stack
EP -> highest stack address this procedure might need

* Note if **EP>NP** (heap bottom) then memory exceeded

Instruction Format

OP: 6 bit instruction code (max 64 codes)
P: 4 bit modifier
often nesting level (max nesting is 16)
P=0 is self; P=1 is parent; etc.
often data type (e.g., used for base of constants)
A: 20 bit address (limited to 1MB / space)
can be stack offset; can be code offset;
can be constant data offset

Stack Frames for Pascal-S

0: return value space

1: return address (old PC value)

2: display ptr (data structure for called routines)

3: previous MP (dynamic link)

4: Called proc id (index to get info on routine);

Used for static linking as well –
no need to follow chain back

5… Zeroed-out for parameters and temps

Pascal Nesting

```
program factor(input,output);
        var f: integer;
        function factorial(f:integer):integer;
                function fact(n:integer):integer;
                begin
                        if n=1 then fact := 1 else fact := n*fact(n-1)
                end; (* end fact *)
        begin
                if f<=0 then factorial := 0 else factorial := fact(f)
        end; (* end factorial *)
begin
        readln(f);
        writeln(factorial(f));
        readln (* just to be able to read results on console *)
end.
```

Pascal-S P-Code #1

```
fact(n:integer):integer (level 3)
0:      LDO     n (3,5)*
1:      LDC     1
2:      EQUAL // if n=1
3:      FJP     8
4:      LDA     fact (3,0)*
5:      LDC     1
6:      STO     // fact := 1
7:      JMP     18
8:      LDA     fact (3,0)*
9:      LDO     n (3,5)*
10:     MST     fact (32)**
11:     LDO     n (3,5)*
12:     LDC     1
13:     SUB
14:     CALL    5***
15:     UPD     2,3****
16:     MPI
17:     STO     // fact:=n*fact(n-1)
18:     Exit
* static link, offset
** space checked. SP=SP+5
*** back to top of frame
**** unwind display after recursion
```

Pascal-S P-Code #2

```
factorial(f:integer):integer (level 2)
19:     LDO     f  (2,5)
20:     LDC     0
21:     LE      // if f<=0
22:     FJP     27
23:     LDA     factorial (2,0)
24:     LDC     0
25:     STO     // factorial := 0
26:     JUMP    32
27:     LDA     factorial (2,0)
28:     MST     fact (32);
29:     LDO     f  (2,5)
30:     CALL    5
31:     STO     // factorial := fact(f)
32:     EXIT;
```

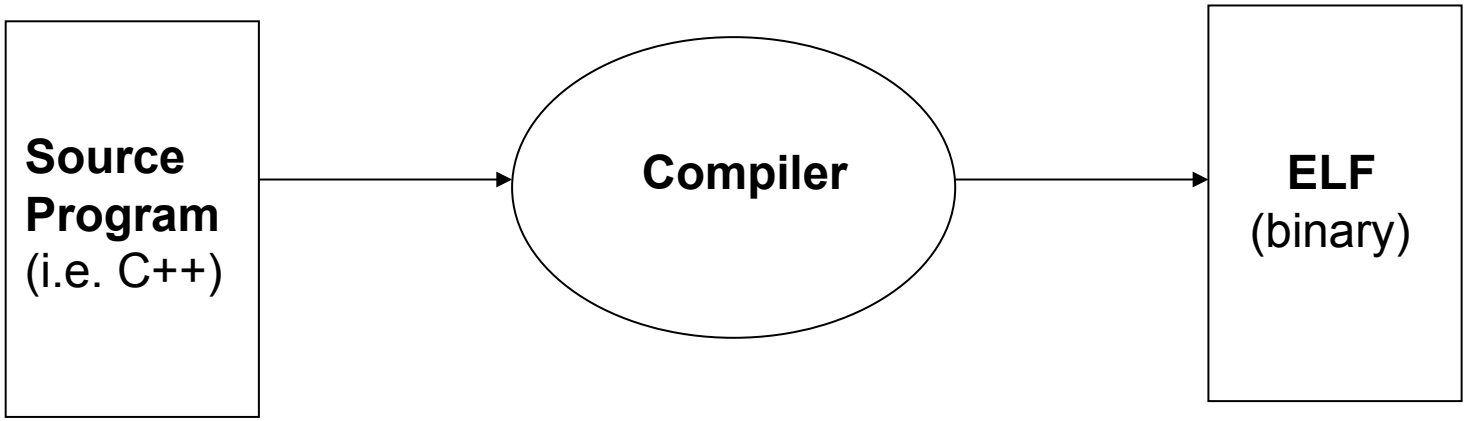
Pascal-S P-Code #3

```
factor(input,output)
33:     LDA         f (1,5);
34:     READINT     // readln(f)
35:     READLN
36:     MST         factorial (30)
37:     LDO         f (1,5)
38:     CALL        5
39:     WRITEINT    // writeln(factorial(f))
40:     WRITELN
41:     READLN      // readln
42:     END
```

LANGUAGE PROCESSORS: COMPILER, INTERPRETERS & ANALYZERS (DAY #4)

Compilers

- A compiler is a program that takes high level languages (e.g. Pascal, C, C++, Ruby, Java, C#) as input, and translates it to a low-level representation which the computer can understand and execute.



ELF: Executable **L**inkable **F**ile

Language Translators

- Programming languages are notations for describing computations to people and to machines.
- Programming languages can be implemented by any of three general methods:

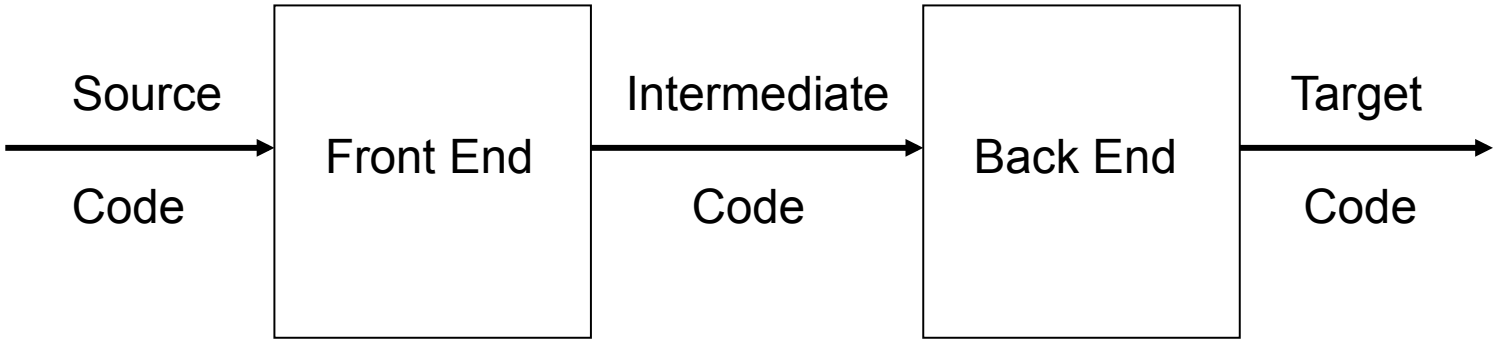
  1. Compilation
  2. Interpretation
  3. Hybrid Implementation (JIT)

Phases of a Compiler

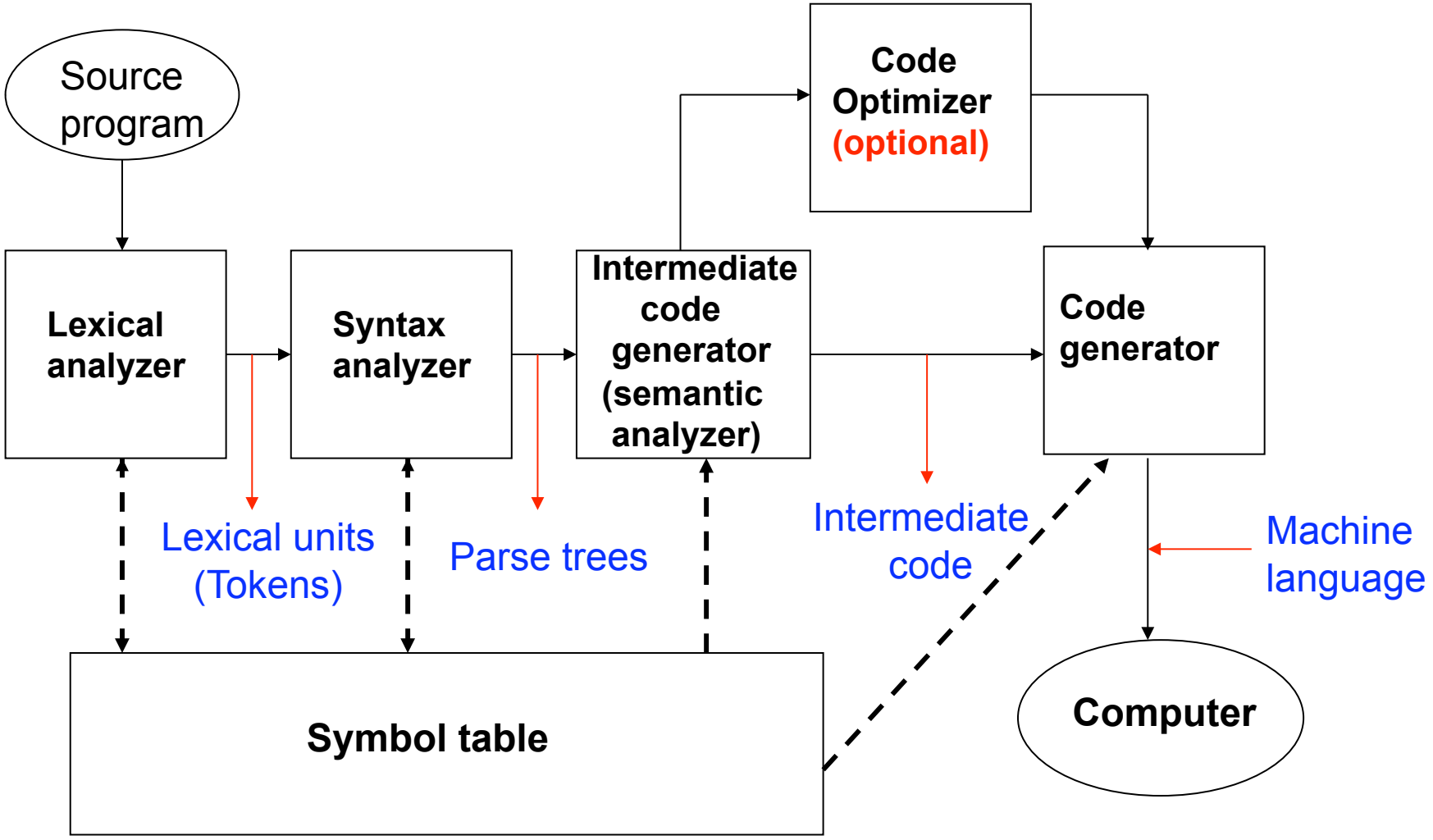
The process of compilation and program execution takes place in several phases:

Front end: Scanner → Parser → Semantic Analyzer

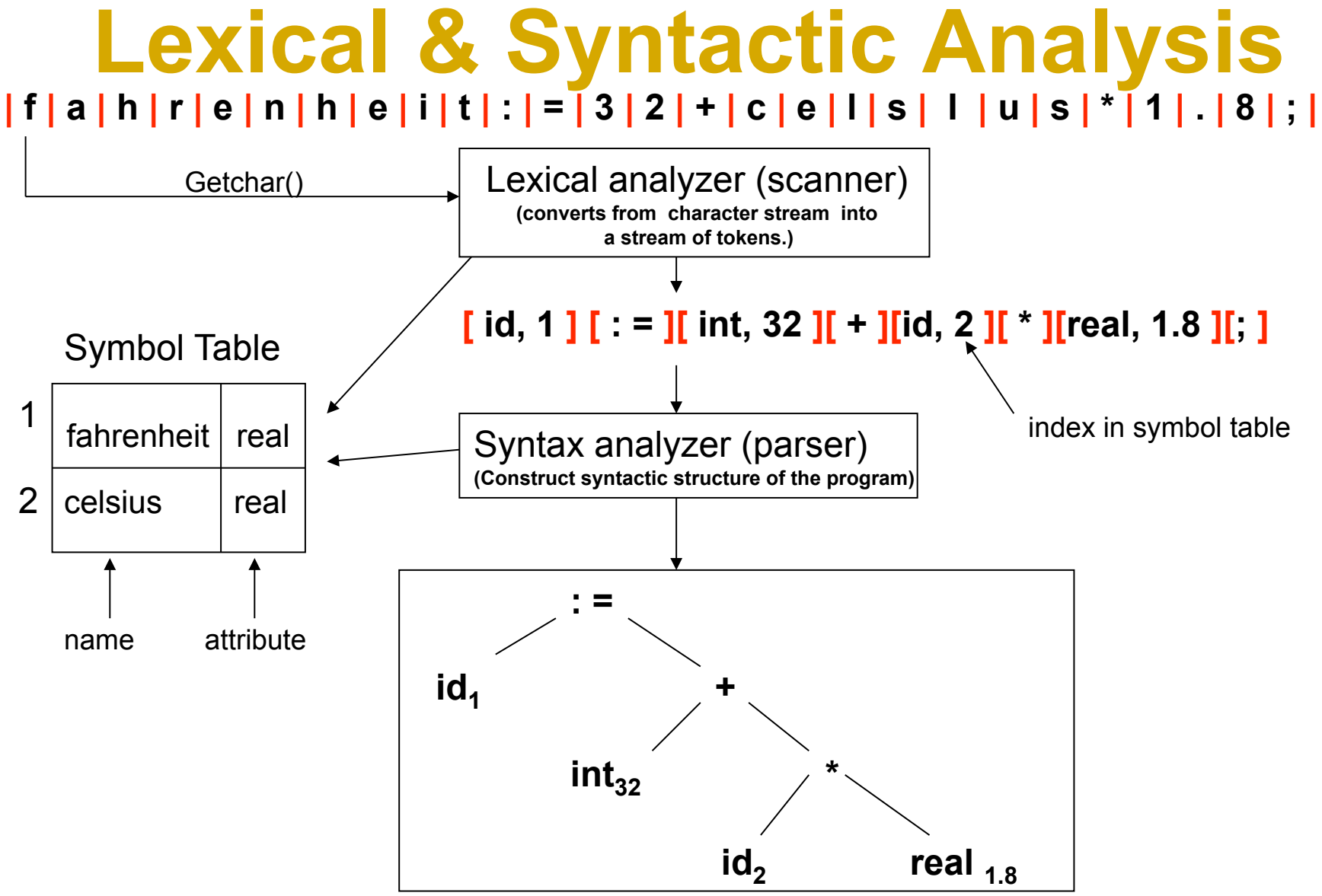
Back end: Code generator



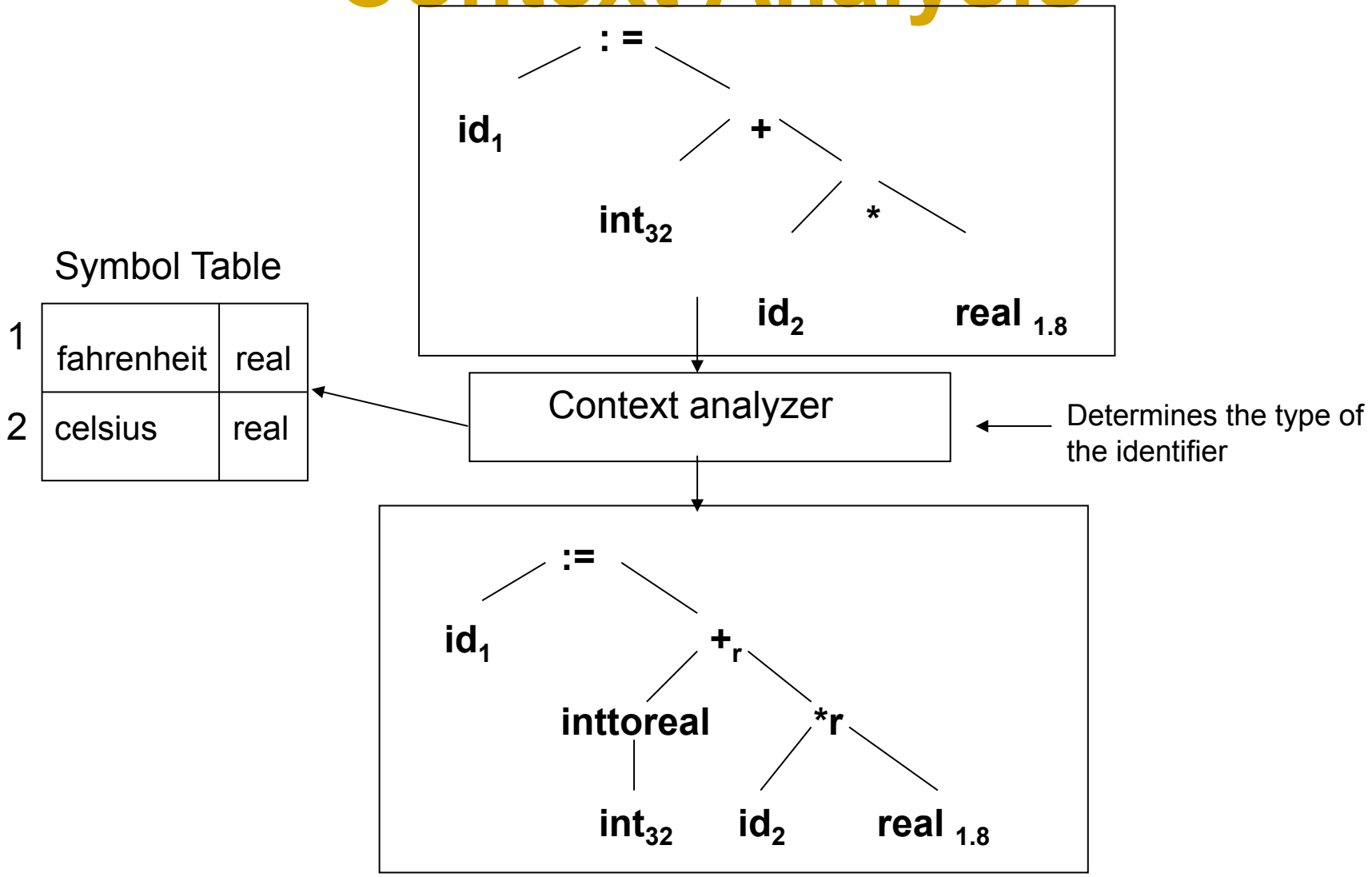
Details on Compiler Process



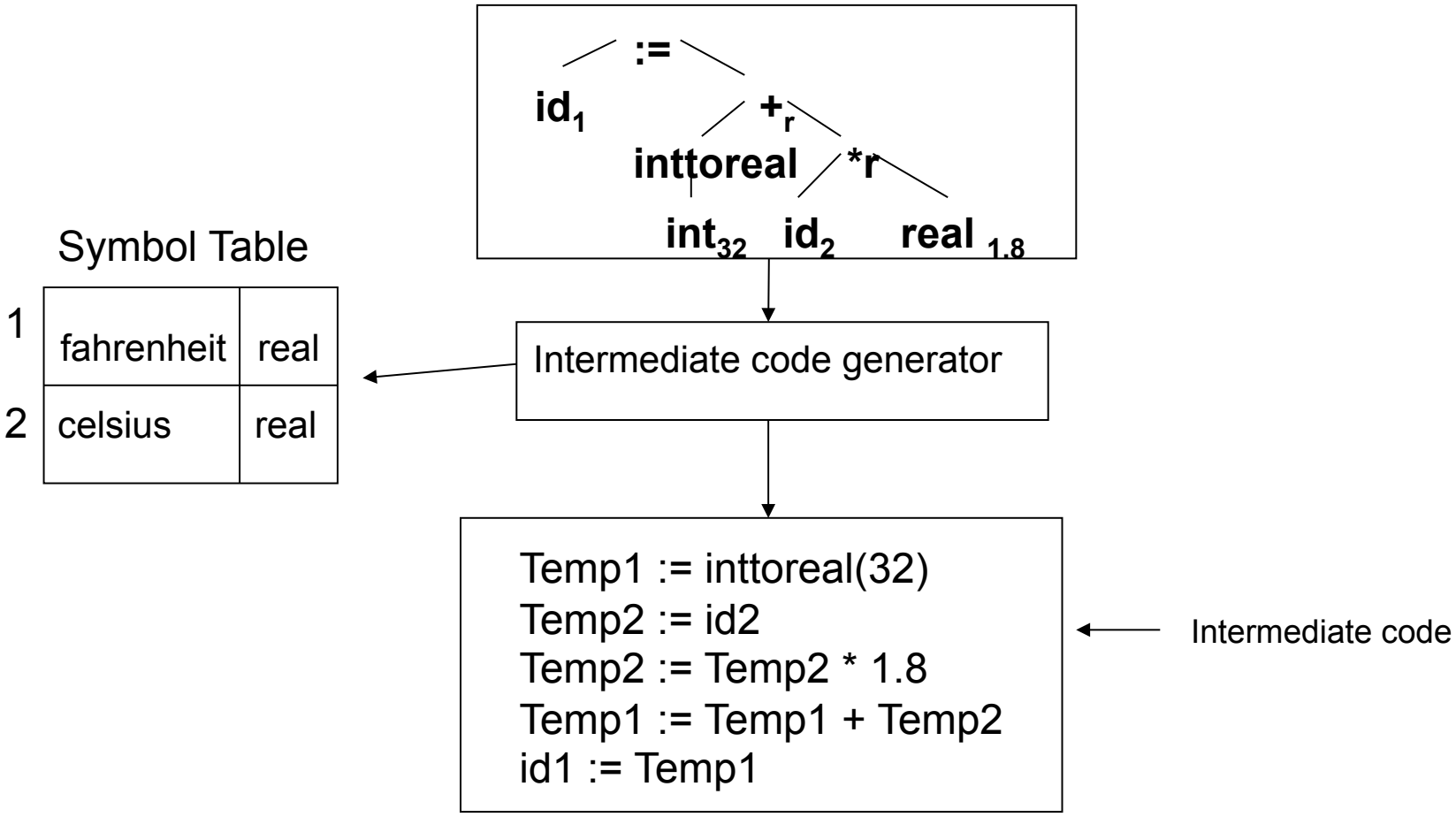
Lexical & Syntactic Analysis



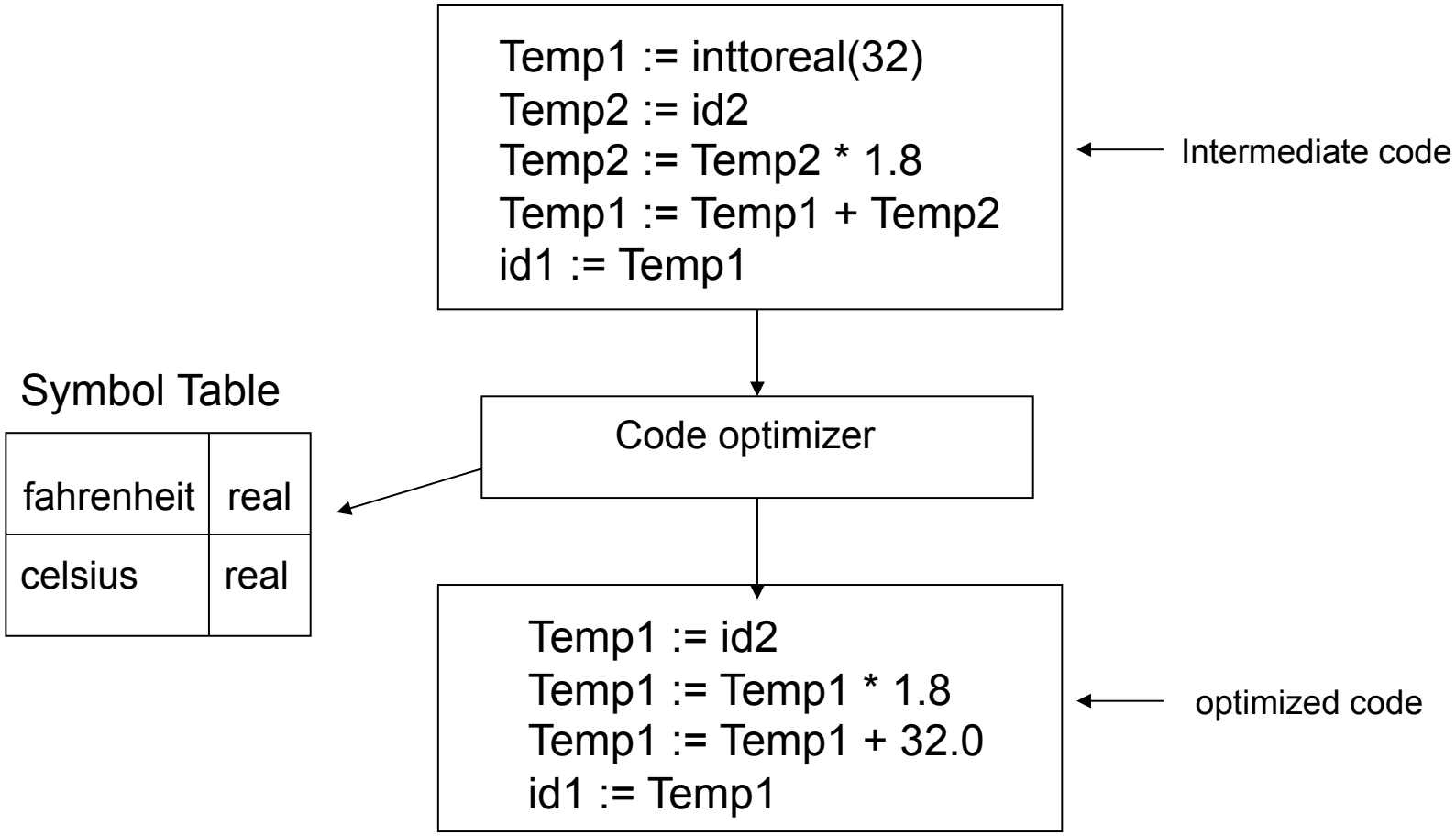
Context Analysis



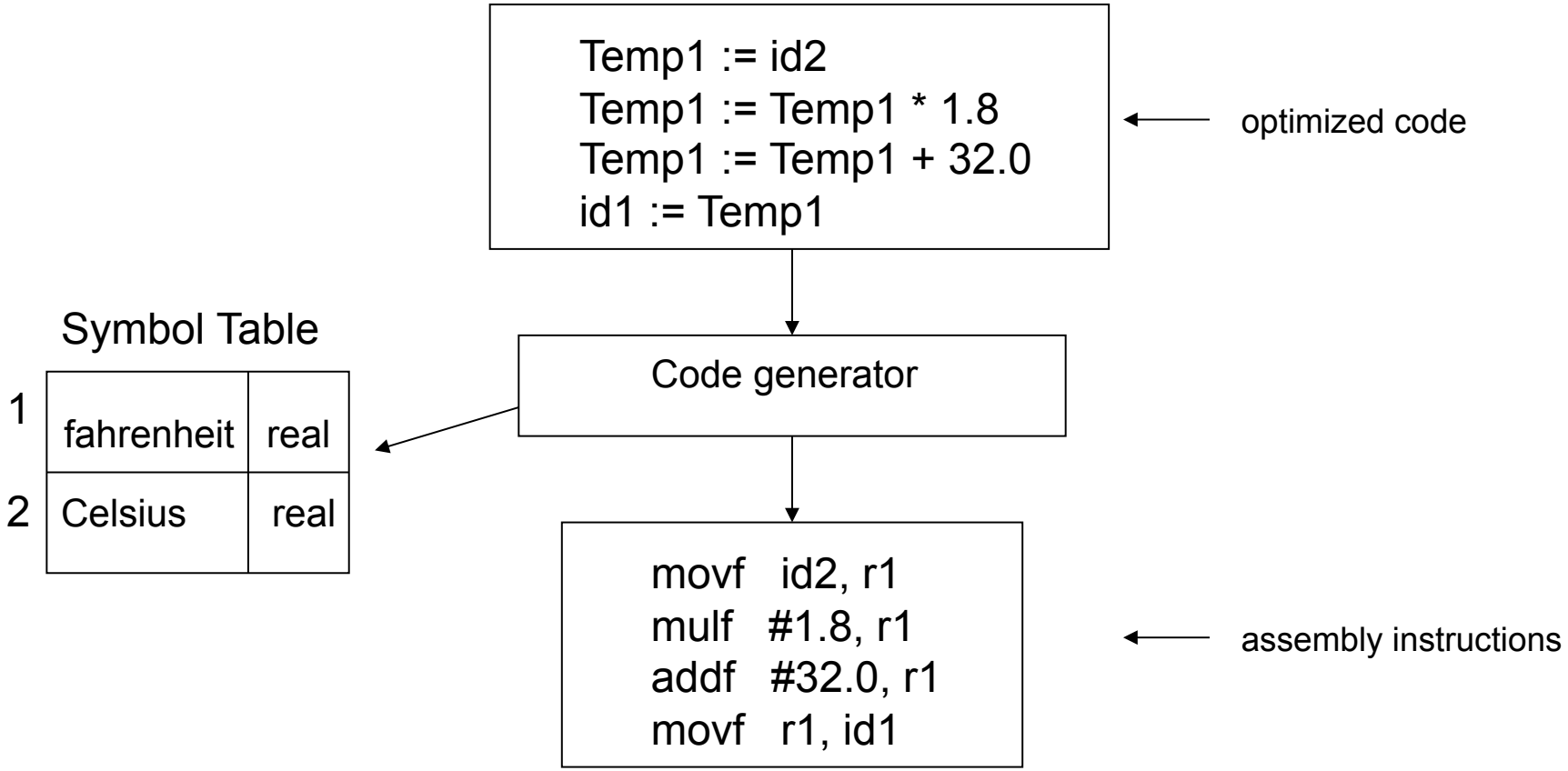
Intermediate Code Gen



Code Improvement



Code Generation



Lexical and Syntactic

Lexical analyzer:

Gathers the characters of the source program into lexical units.

Lexical units of a program are:

- identifiers
- special words (reserved words)
- operators
- special symbols
- Comments are ignored!

Syntax analyzer:

Takes lexical units from the lexical analyzer and use them to construct a hierarchical structure called **parse tree**

- Parse trees represent the syntactic structure of the program.

Interm. Code & Optimization

Intermediate code:

Produces a program in a different language representation:

- Assembly language
- Similar to assembly language
- Something higher than assembly language

Note: semantic analysis is integral part of intermediate code generator

Optimization (really should be called improvement):

- Makes programs smaller or faster or both.
- Most optimization is done on the intermediate code.

Code Gen & Symtab

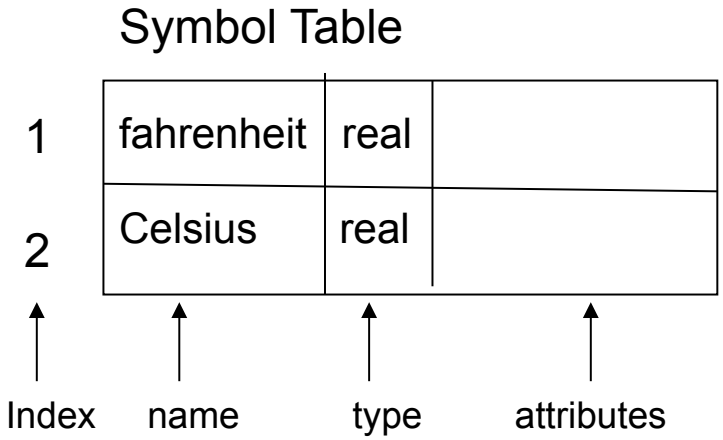
Code generator:

Translate the optimized intermediate code into machine language.

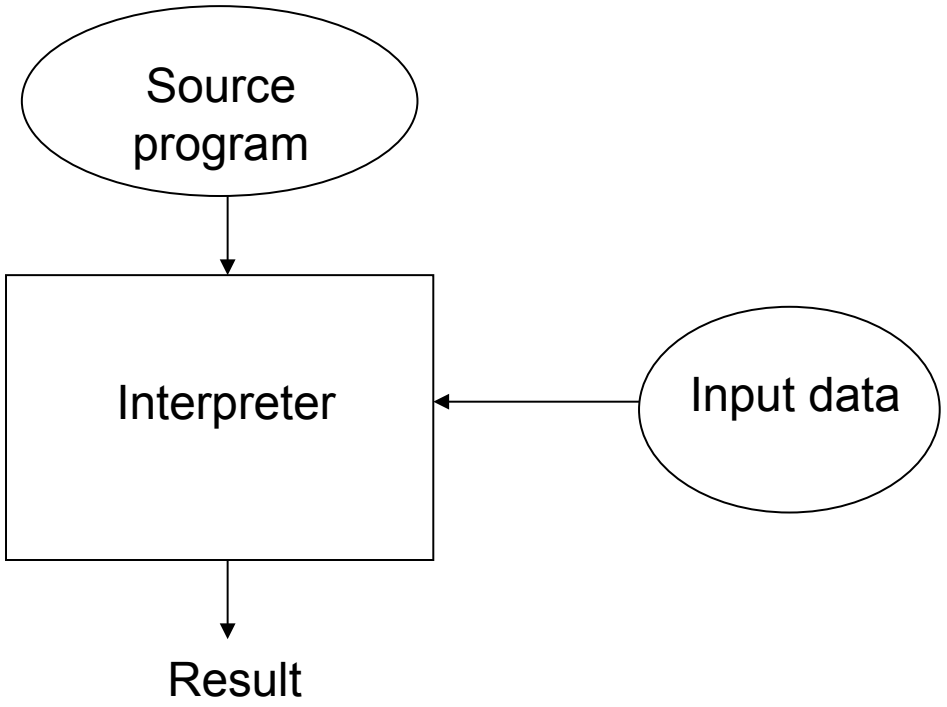
The symbol table:

Serve as a database for the compilation process.

Maintain contents type and attribute information of each user-defined name in the program.



Flow of Interpreter



Interpreters

Programs are interpreted (executed) by another program called the interpreter.

Advantages: Easy implementation of many source-level debugging operations, because all run-time errors operations refer to source-level units.

Disadvantages: 10 to 100 times slower because statements are interpreted each time the statement is executed.

Background:

Early sixties → APL, SNOBOL, Lisp.

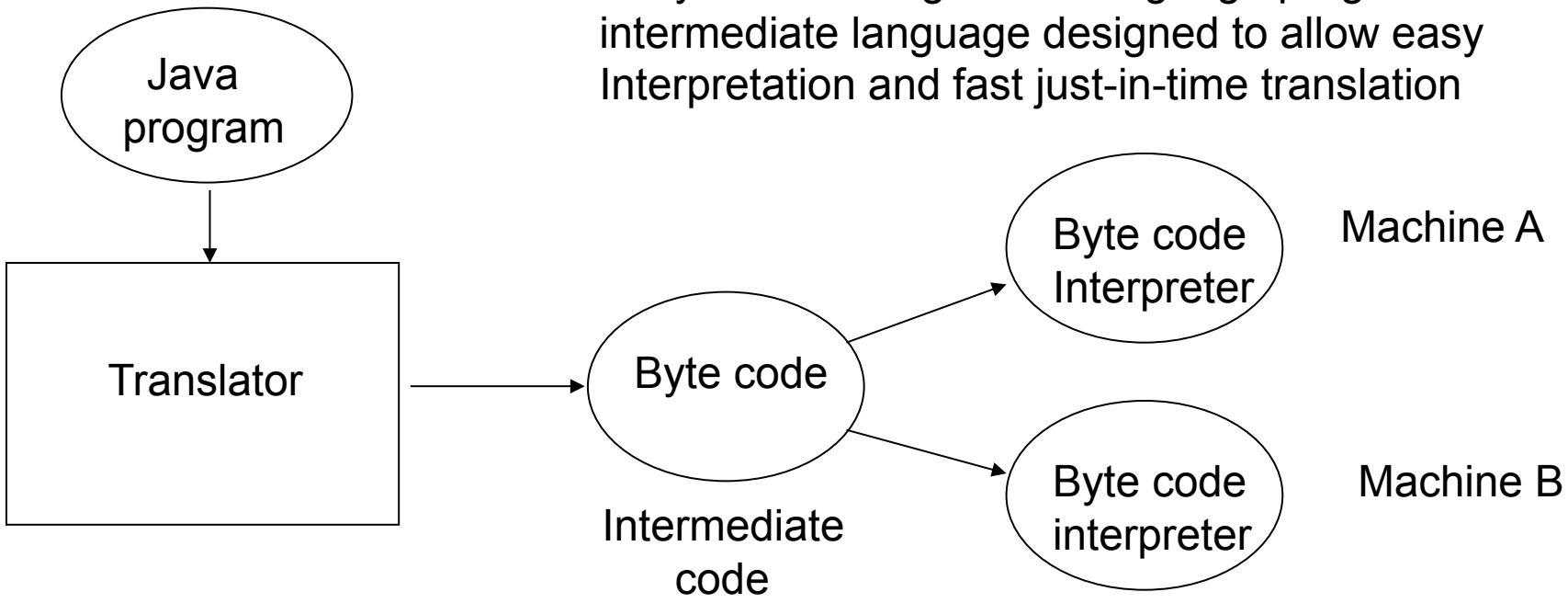
By the 80s → Lisp, Prolog

Recent years → Significant comeback

some Web scripting languages: JavaScript, php

Preparation for Hybrid

They translate high-level language programs to an intermediate language designed to allow easy Interpretation and fast just-in-time translation



Just-in-Time (JIT)

Programs are translated to an intermediate language.

During execution, it compiles intermediate language methods into machine code when they are called (or based on profiling hot spots).

The machine code version is kept for subsequent calls.

.NET and Java programs are implemented with JIT systems.

Pascal-S Language

```
program multiply(input,output);
const m = 7; const n = 85;
var  x,y,z : integer;
procedure mult;
  var a, b : integer;
begin
 a := x;  b := y; z := 0;
 while b > 0 do
   begin
     if odd(b) then z := z+a;
     a := 2*a; b := b div 2;
   end
end;
begin
 x := m; y := n;
 mult;
 writeln(x,'*',y,'=',z); readln
end.
```

As in any language, in Pascal-S we need to identify what is the vocabulary and what are the names and special symbols that we accept as valid.

Reserved words are shown in red.

Operators and special symbols in green

Numeric constants are shown in purple

String constants are shown in gold

Identifiers are shown in black

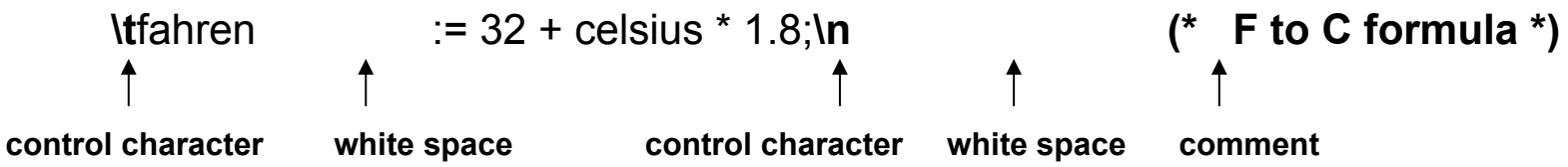
LEXICAL ANALYSIS
Hand carving the Scanner
(DAY #5)

Tasks of Lexical Analysis

1. Read input one character at a time
2. Group characters into tokens
3. Remove white spaces, comments and control characters
4. Encode token types
5. Detect errors and generate error messages

Scanner Example

The stream of characters in the assignment statement



is read in by the Scanner (Lexical Analyzer), which translates it into a stream of tokens in preparation for the Parser (Syntax Analyzer).

[id, fahren] [assign][int, 32][plus][id, celsius][times][real, 1.8][semicolon]

White space (blanks, tabs) are removed. Comments are also not passed along, but they are processed in Scanner if directives can be embedded in them.

1. Lookahead plays an important role in lexical analysis.
2. It is not always possible to decide if a token has been found without looking ahead one character.
3. For instance, if only one character, say “i”, is used it would be impossible to decide whether we are in the presence of identifier “i” or at the beginning of the reserved word “if”.
4. Lookahead is needed for an = in C, as it could be Assign or the start of Equal (==).
5. You must be careful to be consistent. That means always be at the character **after** the token you just transmitted.

Scanner Data Structures

- Define the token types (internal representation)
- Create tables with initial values:
  - Reserved words:
    - **begin**, **const**, **do**, **end**, **if**, **procedure**, **then**, **else**, **while**, etc.
    - Maybe predefined functions and procedures in separate table
  - Special symbols:
    - '**+**', '**-**', '*', '*/*', '**(**', '**)**', '**=**', ',' , '**.**', ' **<**', '>', '**;**'
  - Symbol table (maybe defer to syntax analysis)
  - Constant tables (strings are sometimes done by scanner with all else deferred to parser.
    - String constants are often stored in very compact fashion by recognizing substrings

Scanner and Ordinals

- Scanner must often mess with ordinals of characters. Ordinals are usually expressed in decimal, octal or hex.
- I will discuss the Pascal-S scanner and show you where knowing the ordinals really comes in handy.

ASCII #1

Dec	Hex	ASCII
0	00	NUL (null)
1	01	SOH (start of heading)
2	02	STX (start of text)
3	03	ETX (end of text)
4	04	EOT (end of transmission)
5	05	ENQ (enquiry)
6	06	ACK (acknowledge)
7	07	BEL (bell)
8	08	BS (backspace)
9	09	HT (horizontal tab)
10	0A	LF (line feed)
11	0B	VT (vertical tab)
12	0C	FF (form feed)
13	0D	CR (carriage return)
14	0E	SO (shift out)
15	0F	SI (shift in)

Dec	Hex	ASCII
16	10	DLE (data link escape)
17	11	DC1 (device control 1)
18	12	DC2 (device control 2)
19	13	DC3 (device control 3)
20	14	DC4 (device control 4)
21	15	NAK (negative acknowledge)
22	16	SYN (synchronous idle)
23	17	ETB (end of transmission block)
24	18	CAN (cancel)
25	19	EM (end of medium)
26	1A	SUB (substitute)
27	1B	ESC (escape)
28	1C	FS (file separator)
29	1D	GS (group separator)
30	1E	RS (record separator)
31	1F	US (unit separator)

Dec	Hex	ASCII
32	20	SP (space)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII #2

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
48	30	0	64	40	@	80	50	P
49	31	1	65	41	A	81	51	Q
50	32	2	66	42	B	82	52	R
51	33	3	67	43	C	83	53	S
52	34	4	68	44	D	84	54	T
53	35	5	69	45	E	85	55	U
54	36	6	70	46	F	86	56	V
55	37	7	71	47	G	87	57	W
56	38	8	72	48	H	88	58	X
57	39	9	73	49	I	89	59	Y
58	3A	:	74	4A	J	90	5A	Z
59	3B	;	75	4B	K	91	5B	[
60	3C	<	76	4C	L	92	5C	\
61	3D	=	77	4D	M	93	5D	]
62	3E	>	78	4E	N	94	5E	^
63	3F	?	79	4F	O	95	5F	_

ASCII #3

Dec	Hex	ASCII	Dec	Hex	ASCII
96	60	`	112	70	p
97	61	a	113	71	q
98	62	b	114	72	r
99	63	c	115	73	s
100	64	d	116	74	t
101	65	e	117	75	u
102	66	f	118	76	v
103	67	g	119	77	w
104	68	h	120	78	x
105	69	i	121	79	y
106	6A	j	122	7A	z
107	6B	k	123	7B	{
108	6C	l	124	7C	|
109	6D	m	125	7D	}
110	6E	n	126	7E	~
111	6F	o	127	7F	DEL

Const Declarations

```
program PascalSLex(input, output, srcfil);
(*       N. Wirth, E.T.H CH-8092 Zurich      *)
label 99;               (* escape when input consumed *)

const nkw   =   27;      (* no. of key words *)
      alng  =   10;      (* no. of significant chars in identifiers *)
      blankID = '          '; (* blank string of length alng *)
      llng  =   80;      (* input line length *)
      emax  =  308;      (* max exponent of real numbers *)
      emin  = -324;      (* min exponent *)
      kmax  =   15;      (* max no. of significant digits *)
      smax  =   78;      (* max size of string *)
      ermax =   58;      (* max error no. *)
      nmax  = maxint;    (* largest integer value on this machine *)
      CRLF  = TRUE;      (* true, if palatform uses CR/LF *)
```

Type Declarations

```
type  (* The token types recognized by lex analyzer*)
     symbol = (intcon, realcon, charcon, stringt,
               notsy, plus, minus, times, idiv, rdiv, imod, andsy, orsy,
               egl, neg, gtr, geg, lss, leg,
               lparent, rparent, lbrack, rbrack, comma, semicolon, period,
               colon, becomes, constsy, typesy, varsy, functionsy,
               proceduresy, arraysy, recordsy, programsy, ident,
               beginsy, ifsy, casesy, repeatsy, whilesy, forsy,
               endsy, elsesy, untilsy, ofsy, dosy, tosy, downtosy, thensy);

     (* maximum string needed for identifier or keyword *)
     alfa = packed array [1..alng] of char;
```

VAR Declarations

```
var  sy: symbol;        (* last symbol read by insymbol *)
     id: alfa;          (* identifier from insymbol *)
     inum: integer;     (* integer from insymbol *)
     rnum: real;        (* real number from insymbol *)
     ch: char;          (* last character read from source program *)
     line: array [1..llng] of char;
     cc: integer;       (* character counter *)
     lc: integer;       (* program location counter *)
     ll: integer;       (* length of current line *)
     errs: set of 0..ermax; (* retains list of errors encountered *)
     errpos: integer;   (* error position for lexical error just found*)
     progname: array[1..20] of char; (* input file name *)
     key: array [1..nkw] of alfa; (* set of keywords *)
     ksy: array [1..nkw] of symbol; (* symbols associated with keywords *)
     sps: array [char] of symbol; (* special symbols *)
     st: packed array [0..smax] of char; (* string from insymbol *)
     srcfil: text; { source input file }
     synames: array[symbol] of alfa; (* strings names for symbols *)
```

Next Character

```
procedure nextch; (* read next character; process line end *)
const TAB=9; charPerTab = 8;
begin if cc = ll then
      begin if eof(srcfil) then
        begin writeln; writeln(' source completed'); goto 99 end;
        if errpos <> 0 then begin writeln; errpos := 0  end;
        write(lc:5, '  '); ll := 0; cc := 0; lc := lc+1;
        while not eoln(srcfil) do
          begin read(srcfil, ch);
               if ch >= ' '  then begin ll := ll+1; write(ch); line[ll] := ch end
               else if ord(ch)=TAB then
                    repeat ll:=ll+1; write(' '); line[ll]:=' ' until (ll mod charPerTab )=1
          end;
        writeln; ll := ll+1; read(srcfil, line[ll]); if CRLF then read(srcfil, line[ll])
       end;
   cc := cc+1; ch := line[cc];
end (* nextch *);
```

Errors

```
procedure error(n: integer);  (* position carat (^) under error *)
begin
   if errpos = 0 then write(' ****');
   if cc > errpos then begin
      write(' ': cc-errpos, '^', n:2);
      errpos := cc+3; errs := errs + [n]
   end
end (* error *);
```

Read scale

```
procedure insymbol;   (* reads next symbol *)
  label 1, 2, 3; (* EVIL!!!! *)
  var i, j, k, e: integer;

  procedure readscale;
    var s, sign: integer;
  begin
    nextch; sign := 1; s := 0;
    if ch = '+' then nextch
    else if ch = '-' then begin nextch; sign := -1 end;
    while ch in ['0'..'9'] do begin s := 10*s + ord(ch) - ord('0'); nextch end;
    e := s*sign + e
  end (* readscale *);
```

Adjust scale

```
procedure adjustscale;
  var s: integer; d, t: real;
begin
  if k+e > emax then error(21)
  else if k+e < emin then rnum := 0
  else begin
    s := abs(e); t := 1.0; d := 10.0;
    repeat
      while not odd(s) do begin s := s div 2; d := sqr(d) end;
      s := s-1; t := d*t
    until s = 0;
    if e >= 0 then rnum := rnum*t  else  rnum := rnum/t
  end
end (* adjustscale *);
```

Handle names (ID, Keyword)

```
begin (* insymbol *)
1: while ch <= ' ' do nextch;
  if ch in ['a'..'z'] then begin (* word *)
    k := 0; id := '          '; (* Ugly because requires alng knowledge *)
    repeat
      if k < alng then
        begin k := k+1; if ch in ['A'..'Z'] then ch := chr(ord(ch)+32); id[k] := ch end;
      nextch
    until not (ch in ['A'..'Z', 'a'..'z', '0'..'9']);
    i := 1; j := nkw; (* binary search *)
    repeat
      k := (i+j) div 2; if id <= key[k] then j := k-1; if id >= key[k] then i := k+1
    until i > j;
    if i-1 > j then sy := ksy[k] else sy := ident
  end
```

Handle numbers

```
else if ch in ['0'..'9'] then begin (* number *)
    k := 0; inum := 0; sy := intcon;
    repeat inum := inum*10 + ord(ch) - ord('0'); k := k+1; nextch until not (ch in ['0'..'9']);
    if (k > kmax) or (inum > nmax) then begin error(21); inum := 0; k := 0 end;
    if ch = '.' then begin
      nextch;
      if ch = '.' then ch := ':'
      else begin
        sy := realcon; rnum := inum; e := 0;
        while ch in ['0'..'9'] do
          begin e := e-1; rnum := 10.0*rnum + (ord(ch)-ord('0')); nextch end;
        if ch = 'e' then readscale; if e <> 0 then adjustscale
       end
    end
    else if ch = 'e' then begin
        sy := realcon; rnum := inum; e := 0; readscale; if e <> 0 then adjustscale
    end
  end
```

Colon (:), <, >, period (.)

```
 else case ch of
':': begin nextch;
        if ch = '=' then
          begin sy := becomes; nextch end  (* := *)
          else sy := colon
    end;
'<': begin nextch;
       if ch = '=' then begin sy := leg; nextch end
       else if ch = '>' then begin sy := neg; nextch end else sy := lss
    end;
'>': begin nextch;
        if ch = '=' then begin sy := geg; nextch end else sy := gtr
    end;
'.': begin nextch;
       if ch = '.' then
          begin sy := colon; nextch end (* ellipsis *)
          else sy := period
    end;
```

Quote (‘) – char or string

```
'': begin k := 0;
   2: nextch;
      if ch = '' then begin nextch; if ch <> '' then goto 3 end;
      if k <= smax then begin
         k := k+1; st[k] := ch
      end;
      goto 2;
  3: if k = 1 then
         begin sy := charcon; inum := ord(stab[sx]) end
      else if k = 0 then
         begin error(38); sy := charcon; inum := 0 end
      else
         begin sy := stringt; inum := sx; sleng := k; sx := sx+k end
     end;
```

Left paren or other special

```
'(': begin nextch;
       if ch <> '*' then sy := lparent
       else begin (* comment *)
         nextch;
         repeat
           while ch <> '*' do nextch;
           nextch
         until ch = ')';
         nextch; goto 1
       end
     end;
'+', '-',  '*', '/', ')', '=', ',', '[', ']', '#', '&', ';':
     begin sy := sps[ch]; nextch end;
'$', '%', '@', '\', '~', '{', '}', '^':
     begin error(24); nextch; goto 1 end (* More ugliness *)
  end; (* case ch *)
  write(synames[sy],' ')  (* Tracing output *)
end (* insymbol *);
```

Init – Keywords

```
procedure init;
begin
  key[ 1] := 'and       '; key[ 2] := 'array     ';
  key[ 3] := 'begin     '; key[ 4] := 'case      ';
  key[ 5] := 'const     '; key[ 6] := 'div       ';
  key[ 7] := 'do        '; key[ 8] := 'downto    ';
  key[ 9] := 'else      '; key[10] := 'end       ';
  key[11] := 'for       '; key[12] := 'function  ';
  key[13] := 'if        '; key[14] := 'mod       ';
  key[15] := 'not       '; key[16] := 'of        ';
  key[17] := 'or        '; key[18] := 'procedure ';
  key[19] := 'program   '; key[20] := 'record    ';
  key[21] := 'repeat    '; key[22] := 'then      ';
  key[23] := 'to        '; key[24] := 'type      ';
  key[25] := 'until     '; key[26] := 'var       ';  key[27] := 'while     ';
```

Init – Keywords to Tokens

```
(* Just used in tracing output *)
  ksy[ 1] := andsy;        ksy[ 2] := arraysy;
  ksy[ 3] := beginsy;      ksy[ 4] := casesy;
  ksy[ 5] := constsy;      ksy[ 6] := idiv;
  ksy[ 7] := dosy;         ksy[ 8] := downtosy;
  ksy[ 9] := elsesy;       ksy[10] := endsy;
  ksy[11] := forsy;        ksy[12] := functionsy;
  ksy[13] := ifsy;         ksy[14] := imod;
  ksy[15] := notsy;        ksy[16] := ofsy;
  ksy[17] := orsy;         ksy[18] := proceduresy;
  ksy[19] := programsy;    ksy[20] := recordsy;
  ksy[21] := repeatsy;     ksy[22] := thensy;
  ksy[23] := tosy;         ksy[24] := typesy;
  ksy[25] := untilsy;      ksy[26] := varsy;
  ksy[27] := whilesy;
```

Init – Special Characters

```
(* Special characters *)
  sps['+'] := plus;        sps['-'] := minus;
  sps['*'] := times;       sps['/'] := rdiv;
  sps['('] := lparent;     sps[')'] := rparent;
  sps['='] := egl;         sps[','] := comma;
  sps['['] := lbrack;      sps[']'] := rbrack;
  sps['#'] := neg;         sps['&'] := andsy;
  sps[';'] := semicolon;
```

Init – Token Names

```
synames[andsy] := 'andsy';          synames[arraysy] := 'arraysy';
synames[beginsy] := 'beginsy';      synames[casesy] := 'casesy';
synames[constsy] := 'constsy';      synames[idiv] := 'idiv';
synames[dosy] := 'dosy';            synames[downtosy] := 'downtosy';
synames[elsesy] := 'elsesy';        synames[endsy] := 'downtosy';
synames[forsy] := 'forsy';          synames[functionsy] := 'functionsy';
synames[ifsy] := 'ifsy';            synames[imod] := 'imod';
synames[notsy] := 'notsy';          synames[ofsy] := 'ofsy';
synames[orsy] := 'orsy';            synames[proceduresy] := 'procedursy';
synames[programsy] := 'programsy'; synames[recordsy] := 'recordsy';
synames[repeatsy] := 'repeatsy';    synames[thensy] := 'thensy';
synames[tosy] := 'tosy';            synames[typesy] := 'typesy';
synames[untilsy] := 'untilsy';      synames[varsy] := 'varsy';
synames[whilesy] := 'whilesy';
```

Init – Simple Counters

```
  lc := 0;  (* line count *)
  ll := 0; (* number of characters in current line *)
  cc := 0; (* character position in current line *)
  ch := ' '; (next character *)
  errpos := 0; (position of most recent error *)
  errs := []; (* empty set of errors encountered so far *)
end; (* init *)
```

Main PascalSLex routine

```
begin  (* PascalSLex main program *)
  writeln;
  writeln('Pascal-S compiler/interpreter');

  write('Enter name of file to be compiled: ');
  readln(progname);
  assign(srcfil,progname);
  reset(srcfil);

  init; (* tables and more done here *)
  while true do insymbol;
99:
  readln;
end. (* PascalSLex *)
```

C Version of nextch()

```
void nextch() {     /* read next character; process line end */
    const integer tab = 9; integer charPerTab = 8;
    if (cc == ll) {
        if (eof(srcfil)) {
            output << NL << " program incomplete" << NL; errormsg(); goto L99; }
        if (errpos != 0) { output << NL; errpos = 0; }
        output << format(lc++,5) << "  "; ll = 0; cc = 0;
        while (! eoln(srcfil)) {
            srcfil >> ch;
            if (ch >= ' ')  {output << ch; line[++ll] = ch; }
            else if (ord(ch)==tab)
                do { ll = ll+1; output << ' '; line[ll] = ' '; } while (!((ll % charPerTab ) == 1)); }
        output << NL; srcfil >> line[++ll]; if (CRLF) srcfil >> line[ll]; }
    ch = line[++cc];
}   /* nextch */
```

Keyword Lookup

- Wirth's Pascal-S compiler used binary search across 27 elements. Cost to search is $\log_2(27)$ or 5 iterations. Could have done hash, but there would be no real gain.
- However, symbol table (not built here) is often large and amenable to hash table.

LEXICAL ANALYSIS
Using Regular Expressions
(DAY #5)

Alphabets

An *alphabet* is a finite set of symbols and the Greek letter *sigma* (Σ) is often used to denote it.

For example: **Σ = {0,1}** → **the binary alphabet**

A ***string*** (string = sentence = word) over an alphabet is a finite sequence of symbols drawn from that alphabet.

Alphabet	**Strings**
Σ = {0,1}	**15, 201, 3**
Alphabet	**Strings**
Σ = {a, b, c, …, z}	**while, for, const**

The ***length*** of a string **s**, usually written **| s |**, is the number of occurrences of symbols in s.

For example: If B = **while** the value of | **s** | = **5**

Note: the ***empty string***, denoted ε ***(epsilon)***, is the string of length zero.

| ε | = 0

Note: the empty string is sometimes denoted λ ***(lambda)***.

Languages

A *language* is any countable set of strings over some fixed alphabet.

For example:
Let **L** be the alphabet of letters and **D** be the alphabet of digits:

L = { A, B, …, Z, a, b, …, z} and **D** = {0, 1, 2, 3, …, 8, 9}

Note: **L** and **D** are languages all of whose strings happen to be of length one. Therefore, an equivalent definition is:

L is the language of uppercase and lowercase letters.
D is the language of digits.

Regular Expressions

- Let Σ be a finite alphabet, then
- Φ is an regular expression (re) denoting the set (language) {**a**}. We say L(Φ) = {} to denote this.
- ε (or λ) is an re denoting the language L(ε) = {ε}
- If **a** $\in \Sigma$, then **a** is a re denoting the set (language) {**a**}.
- If **r** and **s** are regular expressions then
  r | s is an re denoting the set L(**r**|**s**) = L(**r**) $\cup$ L(**s**)
  r $\cdot$ **s** is an re denoting the set L(**r**$\cdot$**s**) = L(**r**) $\cdot$ L(**s**)
    Here, $\cdot$ denotes pairwise concatenation, i.e.,
      $A \cdot B = \{ x\,y \mid x \in A, y \in B \}$
  r* is an re denoting the set L(**r***) = L(**r**)*
    Here, * is called the Kleene star operator, where
      $\varepsilon \in A^*$; if $x \in A^*$ and $y \in A$, then $xy \in A^*$
- Precedence is *, $\cdot$, | Parentheses can override this.
- Nothing else is a regular expression over Σ

More on the Kleene Star

- Let R be an arbitrary regular expression,
  - $R^0 = \varepsilon$ Note that $x \cdot \varepsilon = \varepsilon \cdot x = x$
  - $R^1 = R$
  - $R^2 = R \cdot R$, $L(R^2) = \{ xy \mid x \in R, y \in R \}$
  - …
  - $R^n = R^{n-1} \cdot R = R \cdot R^{n-1}$, when $n>0$
  - $R^* = R^0 \mid R^1 \mid R^2 \mid \ldots \mid R^n \mid \ldots$
  - $R+ = R^1 \mid R^2 \mid \ldots \mid R^n \mid \ldots$
  - $R? = R \mid \varepsilon$? Denotes 0 or 1 occurrence

Extensions

- We can specify a sequence of letters that are consecutive in ASCII by showing [x-y] where x is the lowest lexically and y the highest lexically in the desired range.
- Thus, our identifiers can be specified as [a-zA-Z]([a-zA-Z0-9])+
- Note that the | is omitted in this notation when multiple expressions are chosen from. This can only be done inside the square brackets.
- [^letters] means anything not matching any one of these letters. Thus [^0-9] is any non-digit character.
- Slash, as in \c, can be used to indicate the character c when c is one of the special characters, e.g., z\+ is the string z+, whereas z+ is one or more z's.
- Period stands for any character as in .*HUGHES.* is any string with the word HUGHES embedded in it. If you want a ".", quote it or use the escape character "\" in front of it, as in \.

More reg exp Notation

- The concatenation symbol · is often omitted in regular expressions
- Examples
  - **(C|c)(H|h)21 = {CH21,Ch21, cH21, ch21}**
  - **(+ | -)? [0-9]+ = [0-9]+ | + [0-9]+ | - [0-9]+**
    - Above is a signed or unsigned integer constant
    - This use of a sign is rarely used in our lexical analyzers as the meaning of a sign versus a binary operator is more of a syntax issue

Identifiers as re's

- The identifiers in our simple language are alphanumeric and must start with an alphabetic symbol. Can describe as
  - letter(letter | digit)*
    where letter is [A-Za-z]
    and digit is [0-9]
- We can also use a grammar to describe as
  - letter → A | B | … | Z | a | b | … |z
  - digit → 0 | 1 | … | 9
  - id → letter rest
  - rest → letter rest | digit rest | ε

Lexemes, Patterns,Tokens

A **Lexeme** is the sequence of input characters in the source program that matches the pattern for a token (the sequence of input characters that the token represents).

A **Pattern** is a description of the form that the lexemes of a token may take.

A **Token** is the internal representation of a lexeme. Some tokens may consist only of a name (internal representation) while others may also have some associated values (attributes) to give information about a particular instance of a token.

Examples:

Lexeme	Pattern	Token	Attribute
Any identifier	letter(letter | digit)*	idsym	pointer to symbol table
If	if	ifsym	--
>=	< | <= | > | >= | = | <>	relopsym	GE
57	digit+	intcon	57

LEXICAL ANALYSIS
Using Lex
(DAY #6)

Lex (Flex)

- Lex is a program that generates lexical analyzers from regular expressions. Flex is a descendant of Lex.
- Input sections
  %{
  copied to generated code
  %}
  {definitions}
  %%
  {rules}
  %%
  {user routines – copied to generated code}

Definitions (Calc)

```
%{
#include <stdio.h>

int top = 0, intval;
int stack[20], reg[26];

int ord(letter) {
        if (islower(letter)) return (letter - 'a');
        else return (letter - 'A');
}

%}
%%
```

Rules (Calc)

```
\n              { top = 0; }
=[ \t]*[a-zA-Z]  { if (top>0) { reg[ord(yytext[yyleng-1])] = stack[top-1];
                        printf("%d\n", stack[top-1]); } }
[0-9]+          { sscanf(yytext,"%d",&intval); stack[top++] = intval; }
[a-zA-Z]        { stack[top++] = reg[ord(yytext[0])]; }
"+"             { if (top>0) { stack[top-2] += stack[top-1]; top--; } }
"-"             { if (top>0) { stack[top-2] -= stack[top-1]; top--; } }
"*"             { if (top>0) { stack[top-2] *= stack[top-1]; top--; } }
"/"             { if (top>0) { stack[top-2] /= stack[top-1]; top--; } }
[ \t]           ;
.               { printf("error\n"); }
%%
```

User Routines (Calc)

```
int yywrap () {
 return(1);
}

int main( argc, argv )
int argc;
char **argv;
{
  ++argv, --argc; /* skip over program name */
  if ( argc > 0 ) yyin = fopen( argv[0], "r" );
  else yyin = stdin;
  yylex();
  return(0);
}
```

Pascal-S Scanner in Lex

```
%{

#include <stdio.h> // standard i/o
#include “y.tab.h“ // includes tokens typically defined in parser
int line_no = 1; // use for line count

%}
```

Definitions Section

```
A [aA]
B [bB]
C [cC]
D [dD]
E [eE]
F [fF]
G [gG]
…
U [uU]
V [vV]
W [wW]
X [xX]
Y [yY]
Z [zZ]
NQUOTE [^']

%%
```

Rules Section #1

```
{A}{N}{D}   return(ANDSY);
{A}{R}{R}{A}{Y}   return(ARRAYSY);
{B}{E}{G}{I}{N}   return(BEGINSY);
{C}{A}{S}{E}   return(CASESY);
{C}{O}{N}{S}{T}   return(CONSTSY);
{D}{I}{V}   return(IDIV);
{D}{O}    return(DOSY);
{D}{O}{W}{N}{T}{O}  return(DOWNTOSY);
{E}{L}{S}{E}   return(ELSESY);
{E}{N}{D}   return(ENDSY);
{F}{O}{R}   return(FORSY);
{F}{U}{N}{C}{T}{I}{O}{N} return(FUNCTIONSY);
{I}{F}    return(IFSY);
```

Rules Section #2

```
{M}{O}{D}   return(IMOD);
{N}{O}{T}   return(NOTSY);
{O}{F}    return(OFSY);
{O}{R}    return(ORSY);
{P}{R}{O}{C}{E}{D}{U}{R}{E} return(PROCEDURESY);
{P}{R}{O}{G}{R}{A}{M}  return(PROGRAMSY);
{R}{E}{C}{O}{R}{D}  return(RECORDSY);
{R}{E}{P}{E}{A}{T}  return(REPEATSY);
{T}{H}{E}{N}   return(THENSY);
{T}{O}    return(TOSY);
{T}{Y}{P}{E}   return(TYPESY);
{U}{N}{T}{I}{L}   return(UNTILSY);
{V}{A}{R}   return(VARSY);
{W}{H}{I}{L}{E}   return(WHILESY);

[a-zA-Z]([a-zA-Z0-9])*  return(IDENT);
```

Rules Section #3

```
"="    return(EGL);
">="    return(GEG);
">"    return(GTR);
"<="    return(LEG);
"<"    return(LSS);
"<>"    return(NEG);
“#”    return(NEG);
"+"    return(PLUS);
"-"    return(MINUS);
"/"    return(RDIV);
"*"    return(TIMES);
"["    return(LBRACK);
"]"    return(RBRACK);
"("    return(LPAREN);
")"    return(RPAREN);
```

Rules Section #4

```
":="    return(BECOMES);
".."    return(COLON);
":"     return(COLON); // extension over what Wirth handles in Pascal-S
","     return(COMMA);
"."     return(DOT);
";"     return(SEMICOLON);

'({NQUOTE}|'')+'  return(STRINGT);
([0-9])+    return(INTCON);
/* How would you handle exponents? Can even occur with no decimal point!! */
([0-9])+".“([0-9])*   return(REALCON);

[ \t\f]    ;
\n  line_no++;
```

Rules Section #5

```
“{" { register int c;
      while ((c = input())) {
        if (c == '}') break;
        else if (c == '\n') line_no++;
        else if (c == 0) commenteof();
       }
    }
"(*" { register int c;
      while ((c = input())) {
        if (c == '*') {
         if ((c = input()) == ')') break; else unput (c);
        }
        else if (c == '\n') line_no++;
        else if (c == 0) commenteof();
       }
    }
.   fprintf (stderr, "'%c' (0%o): illegal character at line %d\n", yytext[0], yytext[0], line_no);
%%
```

User Section

```
void commenteof() {
 fprintf (stderr, "unexpected EOF inside comment at line %d\n", line_no);
 exit (1);
}
int yywrap () {
 return (1);
}
int main( argc, argv )
int argc; char **argv;
{ ++argv, --argc; /* skip over program name */
  if ( argc > 0 ) yyin = fopen( argv[0], "r" ); else yyin = stdin;
  yylex();
  return(0);
}
```

Altered for Test (pascal.lex)

```
%%

{A}{N}{D}   printf("ANDSY ");
{A}{R}{R}{A}{Y}   printf("ARRAYSY ");
{C}{A}{S}{E}   printf("CASESY ");
…
[a-zA-Z]([a-zA-Z0-9])*  printf("IDENT %s ", yytext);
…
\'({NQUOTE})*\'  printf("STRINGT %s ", yytext);
([0-9])+\.([0-9])*   printf("REALCON %s ", yytext);
([0-9])+    printf("INTCON %s ", yytext);
[ \t\f]    ;
\n         line_no++; printf("\n%5d ", line_no);
…
  if ( argc > 0 ) yyin = fopen( argv[0], "r" );
  else yyin = stdin;
  printf("\n%5d ", line_no);
```

LEXICAL ANALYSIS
Finite State Automata
(DAY #6)

Transition Diagrams

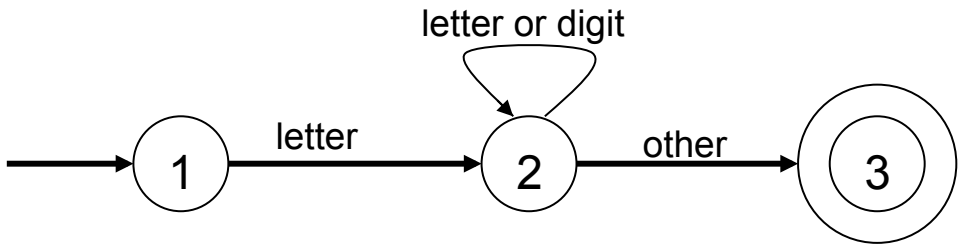
Transition diagrams or transition graphs are used to match a lexeme to a pattern.

Each Transition diagram has:

States	→ represented by circles.
Actions	→ represented by arrows between the states.
Start state	→ represented by an arrowhead (beginning of a pattern)
Final state	→ represented by two concentric circles (end of pattern).

All transition diagrams are deterministic, which means that there is no need to choose between two different actions for a given input.

Example:



ID and Number Diagram

The following state diagrams recognize identifiers and numbers (integers)

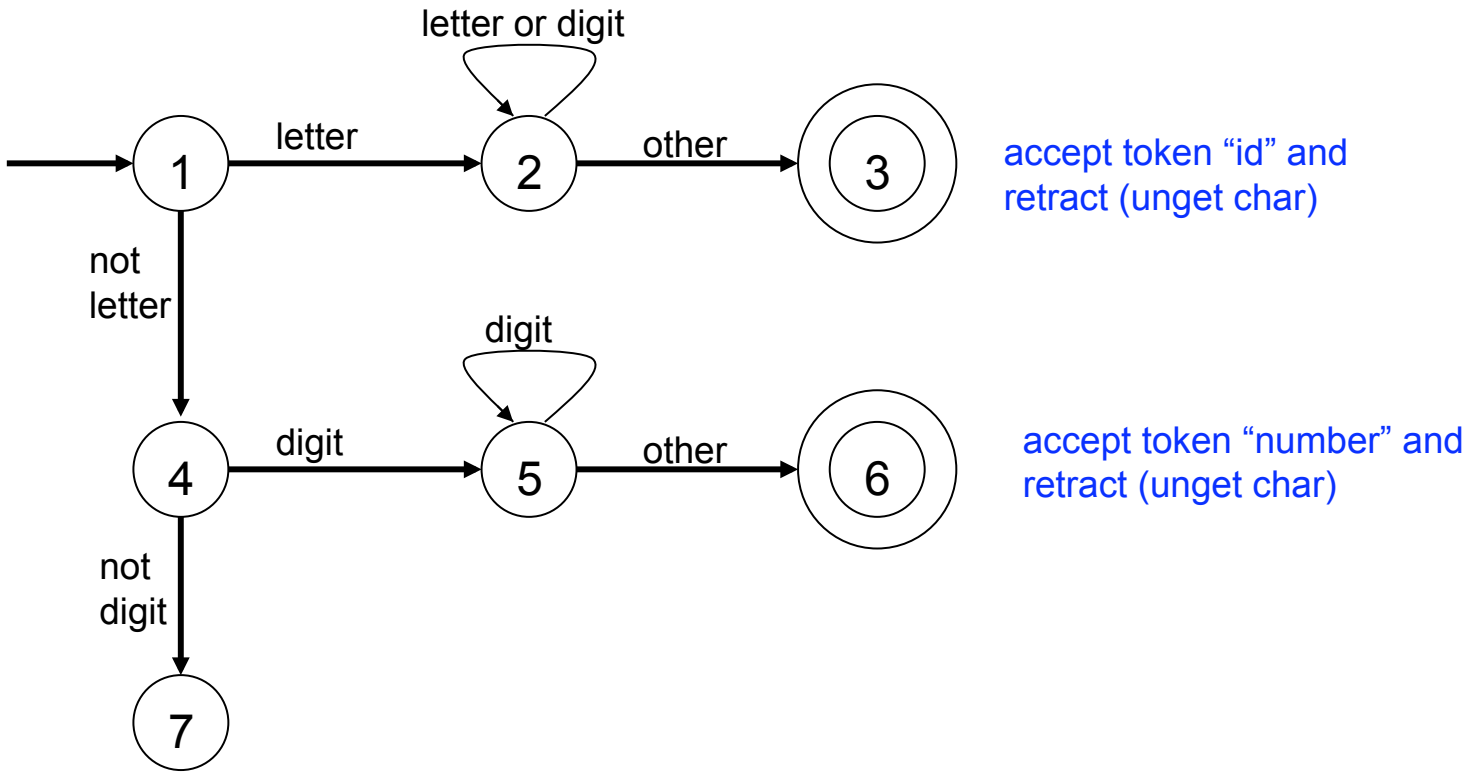


Diagram to Code

Translating transition diagrams to code:

```
{state 1}   ch = getchar();
            If isletter (ch) {

{state 2}     name = (string)ch;
              while (isletter(ch) or isdigit(ch)) do{
                name = concat(name, string(ch));
                ch = getchar();
              }

{state 3}     retract(ch); // we have scanned
                           //  one character too far
              token = (id, name);
              return(token);
            }
```

```
{state 4}   else if isdigit(ch) {
              value := ord(ch)-ord('0');

{state 5}     ch = getchar();
              while (isdigit (ch)) do{
                value := 10 * value + ord(ch)-ord('0')
                ch := getchar
              }

{state 6}     retract(ch);
              token:= (int, value);
              return (token);
            }

{state 7}   else {
              …
            }
```

Finite State Automata (FSA)

- Formal Model for diagrams
  - Finite number of states
  - Transitions based on next character read
  - One (or more) start states
  - Usually many final states
  - Every re has an associated FSA
  - Every FSA has an associated re

Deterministic FSA

- If every state/character combination is associated with precisely one transition then the FSA is deterministic.
- If there are transitions on ε or there is at least one state/character combination for which there is not precisely one transition then the FSA is non-deterministic.

Pattern Matching

- GREP – pattern matching from Unix
- Covering an re to an FSA naturally leads to a non-deterministic one (NFA)
- Converting to a det. FSA (DFA) leads to state explosion (can be exponential)
- DFA runs in linear time; NFA requires backtrack
- Convert if can amortize cost of conversion
- Can amortize in a lexical analyzer because it's run over and over again; can amortize if looking for patterns in a large corpus of text; cannot amortize on trivial scan

CONTEXT FREE GRAMMARS (DAY #7)

Parsing

Regular language nested structures cannot be expressed.

Nested structures can be expressed with the aid of recursion.

For example, a FSA cannot suffice for the recognition of sentences in the set

$$\{ \mathbf{a}^n \mathbf{b}^n \mid n \text{ is in } \{ 0, 1, 2, 3, \ldots\}\}$$

where **a** represents “(“ or “{“

and **b** represents “)” or “}”

Regular + Recursion

So far we have been working with three rules to define regular sets (regular languages):

Concatenation → (s r)

Alternation (choice) → (s | r)

Kleene closure (repetition) → (s)*

Regular sets are generated by regular expressions and recognized by scanners (FSA).

By adding recursion as an additional rule we can define context free languages.

Context Free

Any set of strings that can be defined using concatenation, alternation, Kleene closure and recursion is called a Context Free Language (CFL).

CFLs are generated by Context Free Grammars (CFG) and can be recognized by Pushdown Automatas.

"**Every language has a structure called its grammar**"

Parsing is the task of determining the structure or syntax of a program.

Simple Example of Grammar

Let us observe the following three rules (grammar):

1)<sentence> → <subject> <predicate>

where "→" means "is defined as" or "derives"

2)<subject> → **John** | **Mary**

3)<predicate> → **eats** | **talks**

where " | " is called alternation and means "or"

With these rules we define four possible sentences:

John eats **John talks** **Mary eats** **Mary talks**

Another Simple Grammar

We will refer to the formulae or rules used in the former example as

Syntax rules, productions, syntactic equations, or rewriting rules.

<subject> and <predicate> are syntactic classes or categories, also called non-terminals.

Using a shorthand notation we can write the following syntax rules

S → A B

A → a | b

B → c | d

S is the start symbol

L = { ac, ad, bc, bd} = set of sentences

L is called the language that can be generated From the syntax rules by repeated substitution

History of Formal Language

- In 1940s, Emil Post (mathematician) devised rewriting systems as a way to describe how mathematicians do proofs. Purpose was to mechanize them.
- Early 1950s, Noam Chomsky (linguist) developed a hierarchy of rewriting systems (grammars) to describe natural languages.
- Late 1950s, Backus-Naur (computer scientists) devised BNF (a variant of Chomsky's context-free grammars) to describe the programming language Algol.
- 1960s was the time of many advances in parsing. In particular, parsing of context free was shown to be no worse than $O(n^3)$. More importantly, useful subsets were found that could be parsed in O(n).
- Will discuss the issues faced in 1960s in much more detail as we go along.

Formalism for Grammars

Definition : A language is a set of strings of characters from some alphabet.

The strings of the language are called sentences or statements.

A string over some alphabet is a finite sequence of symbols drawn from that alphabet.

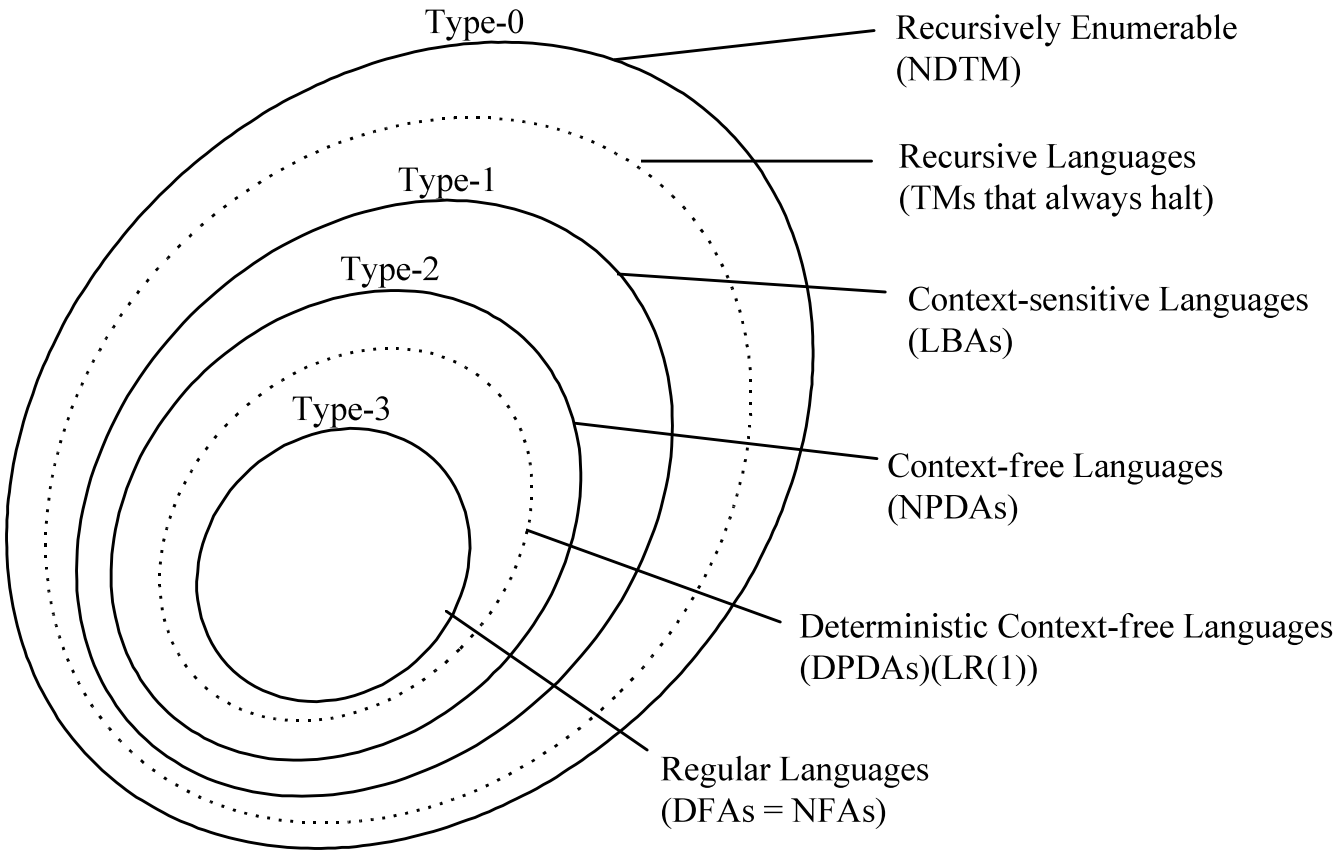
A meta-language is a language that is used to describe another language.

A very well known meta-language is BNF (Backus Naur Form)

It was developed by John Backus and Peter Naur, in the late 50s, to describe programming languages.

Noam Chomsky in the early 50s developed context free grammars which can be expressed using BNF.

Languages – The Big Picture



Context Free Grammars

G = (V, Σ, S, P) where

V is a finite set of symbols called the non-terminals or variables. They are not part of the language generated by the grammar.

Σ is a finite set of symbols, disjoint from V, called the terminals. Strings in the language are made up entirely of terminal symbols.

S is a member of V and is called the start symbol.

P is a finite set of rules or productions. Each member of P is one the form

$A \rightarrow \alpha$ where α is a strings $(V \cup \Sigma)^*$

Note that the left hand side of a rule is a letter in V;

The right hand side is a string from the combined alphabets

The right hand side can even be empty (ε)

Interesting Sample CFG

Example of a grammar for a small language:

G = ({<program>, <stmt-list>, <stmt>, <expression>},
{begin, end, ident, ;, =, +, -}, <program>, P) where P is

<program> → begin <stmt-list> end

<stmt-list> → <stmt> | <stmt> ; <stmt-list>

<stmt> → ident = <expression>

<expression> → ident + ident | ident - ident | ident

Here "ident" is a token return from a scanner, as are "begin", "end", ";", "=", "+", "-"

Note that ";" is a separator (Pascal style) not a terminator (C style).

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1 <assgn> → <id> := <expr>
R2 <id> → a | b | c
R3 <expr> → <id> + <expr>
R4 | <id> * <expr>
R5 | (<expr>)
R6 | <id>

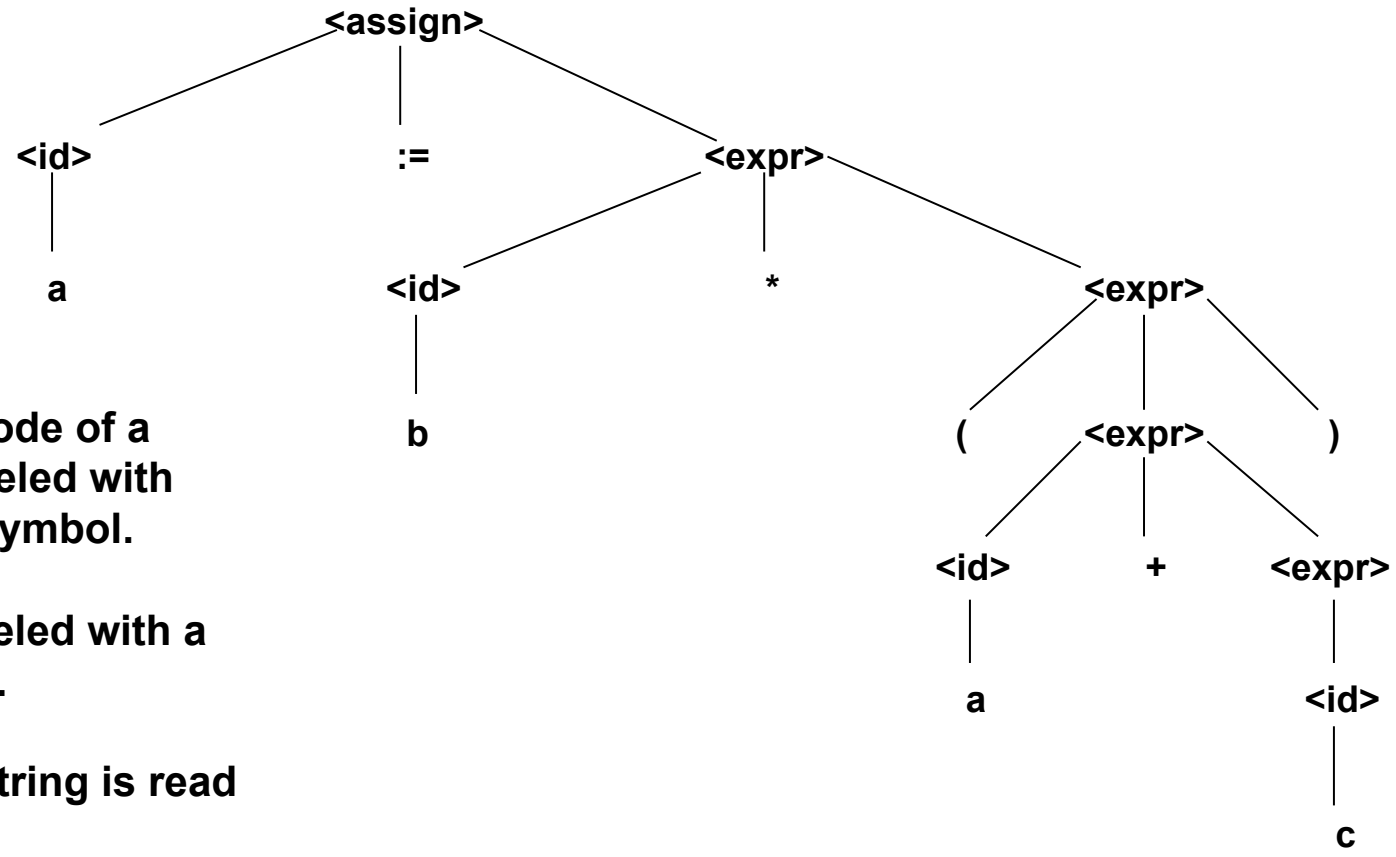
In a **left most derivation** only the left most non-terminal is replaced

The statement a := b * (a + c)
Is generated by the left most derivation:

<assgn> ⇒ <id> := <expr> R1
⇒ a := <expr> R2
⇒ a := <id> * <expr> R4
⇒ a := b * <expr> R2
⇒ a := b * (<expr>) R5
⇒ a := b * (<id> + <expr>) R3
⇒ a := b * (a + <expr>) R2
⇒ a := b * (a + <id>) R6
⇒ a := b * (a + c) R2

Parse Trees

A parse tree is a graphical representation of a derivation
For instance the parse tree for the statement a := b * (a + c) is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

The generated string is read left to right

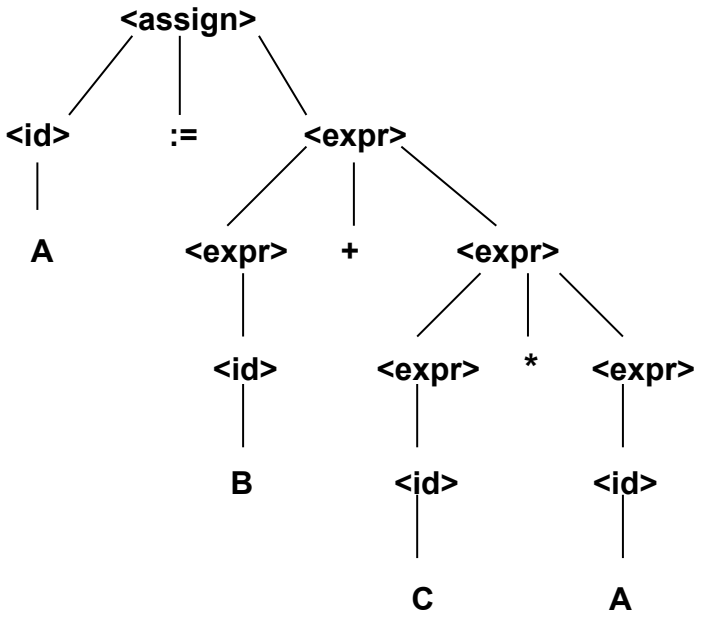
Ambiguity

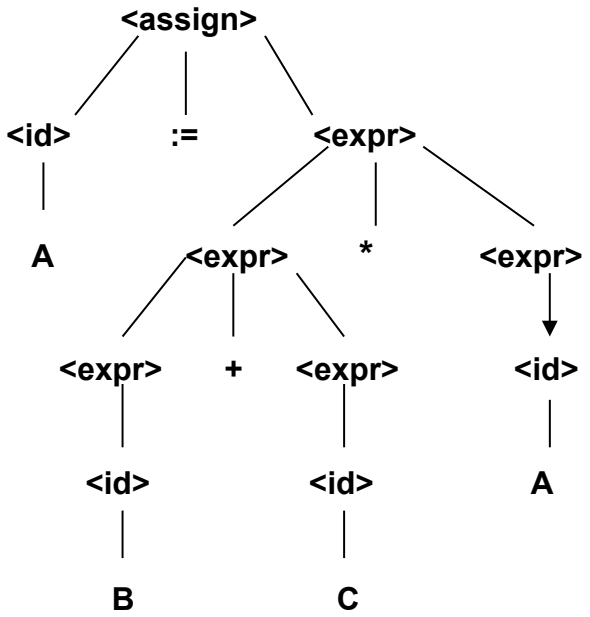
A grammar that generates a sentence for which there are two or more distinct parse trees is said to be "ambiguous"

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression a := b + c * a

<assgn> → <id> := <expr>
<id> → a | b | c
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Ambiguous Parse





This grammar generates two parse trees for the same expression.

**If a language structure has more than one parse tree,
the meaning of the structure cannot be determined uniquely.**

Precedence

Operator precedence:

If an operator is generated lower in the parse tree, it indicates that the operator has precedence over the operator generated higher up in the tree.

An unambiguous grammar for expressions:

```
<assign> → <id> := <expr>
<id>     → a | b | c
<expr>   → <expr> + <term>
         |  <term>
<term>   → <term> * <factor>
         |  <factor>
<factor> →  ( <expr> )
         | <id>
```

This grammar indicates the usual precedence order of multiplication and addition operators.

This grammar generates unique parse trees independently of doing a rightmost or leftmost derivation

Left (right)most Derivations

Leftmost derivation:

<assgn> → <id> := <expr>
→ a := <expr>
→ a := <expr> + <term>
→ a := <term> + <term>
→ a := <factor> + <term>
→ a := <id> + <term>
→ a := b + <term>
→ a := b + <term> *<factor>
→ a := b + <factor> * <factor>
→ a := b + <id> * <factor>
→ a := b + c * <factor>
→ a := b + c * <id>
→ a := b + c * a

Rightmost derivation:

<assgn> ⇒ <id> := <expr>
⇒ <id> := <expr> + <term>
⇒ <id> := <expr> + <term> *<factor>
⇒ <id> := <expr> + <term> *<id>
⇒ <id> := <expr> + <term> * a
⇒ <id> := <expr> + <factor> * a
⇒ <id> := <expr> + <id> * a
⇒ <id> := <expr> + c * a
⇒ <id> := <term> + c * a
⇒ <id> := <factor> + c * a
⇒ <id> := <id> + c * a
⇒ <id> := b + c * a
⇒ a := b + c * a

Avoiding Ambiguity

Dealing with ambiguity:

Rule 1: * (times) and / (divide) have higher precedence than + (plus) and – (minus).

Example:

a + c * 3 ➔ a + (c * 3)

Rule 2: Operators of equal precedence associate to the left.

Example:

a + c + 3 ➔ (a + c) + 3

Unambiguous Grammar

Rewrite the grammar to avoid ambiguity.

The grammar:

<expr> → <expr> <op> <expr> | id | int | (<expr>)
<op> → + | - | * | /

Can be rewritten it as:

<expr> → <term> | <expr> + <term> | <expr> - <term>
<term> → <factor> | <term> * <factor> | <term> / <factor>.
<factor> → id | int | (<expr>)

RECURSIVE DESCENT PARSING (DAY #8,9)

Parsing Problem

The parsing Problem: Take a string of symbols in a language (tokens) and a grammar for that language to construct the parse tree or report that the sentence is syntactically incorrect.

For correct strings:

Sentence + grammar → parse tree

For a compiler, a sentence is a program:

Program + grammar → parse tree

Types of parsers:

Top-down aka predictive (recursive descent parsing)

Bottom-up parsing.

"We will focus on top-down parsing at present".

Top Down Parsing

Recursive Descent parsing uses recursive procedures to model the parse tree to be constructed. The parse tree is built from the top down, trying to construct a left-most derivation.

Beginning with ***start*** symbol, for each non-terminal (syntactic class) in the grammar a procedure which parses that syntactic class is constructed.

Consider the expression grammar:

E → T E'

E' → **+** T E' | **e**

T → F T'

T' → ***** F T' | **e**

F → **(** E **)** | **id**

The following procedures have to be written:

Recursive Descent

```
Procedure E
  begin { E }
    call T
    call E'
    print (" E found ")
  end { E }

Procedure E'
  begin { E' }
    If token = "+" then
      begin { IF }
        print (" + found ")
        Get next token
        call T
        call E'
      end { IF }
      print (" E' found ")
  end { E' }
```

```
Procedure T
  begin { T }
    call F
    call T'
    print (" T found ")
  end { T }

Procedure T'
  begin { T' }
    If token = " * " then
      begin { IF }
        print (" * found ")
        Get next token
        call F
        call T'
      end { IF }
      print (" T' found ")
  end { T' }
```

```
Procedure F
  begin { F }
    case token is
    "(":
        print (" (  found ")
        Get next token
        call E
        if token = ")" then
          begin { IF }
            print (" ) found")
            Get next token
             print (" F found ")
          end { IF }
        else
        call ERROR
    "id":
        print (" id found ")
        Get next token
        print (" F found ")
     otherwise:
        call ERROR
  end { F }
```

Left Recursion & Top-Down

Ambiguity is not the only problem associated with recursive descent parsing.
Other problems to be aware of are left recursion and left factoring:

Left recursion: A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \rightarrow A\ \alpha$ for some non-empty string α.

A is left-recursive if the left-most symbol in any of its alternatives either immediately (direct left-recursive) or through other non-terminal definitions (indirect/hidden left-recursive) rewrites to a string with A on the left.

Top-down parsing methods cannot handle left-recursive grammars,
so a transformation is needed to eliminate left recursion.

Prediction and Left Recursion

Immediate left-recursion: A → A α

E.g., Expr → Expr + Term

Top-down parser implementation:

```
function expr() {
    expr(); match('+'); term();
}
```

Do you see the problem ?

Indirect left-recursion: A → Ba | C

B → Ab | D

A ⇒ Ba ⇒ Aba

Removing Left Recursion

Given left recursive and non left recursive rules

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

Can view as

$A \rightarrow (\beta_1 \mid \ldots \mid \beta_m)\ (\alpha_1 \mid \ldots \mid \alpha_n\ )^*$

Star notation is an extension to normal notation with obvious meaning

Now, it should be clear this can be done right recursive as

$A \rightarrow \beta_1 \mid \ldots \mid \beta_m\ B$

$B \rightarrow \alpha_1 B \mid \ldots \mid \alpha_n B \mid \varepsilon$

Right Recursive Expressions

Grammar: Expr → Expr + Term | Term
Term → Term * Factor | Factor
Factor → (Expr) | Int

Fix: Expr → Term ExprRest
ExprRest → + Term ExprRest | ε
Term → Factor TermRest
TermRest → * Factor TermRest | ε
Factor → (Expr) | Int

This is syntactically fine, but semantically it can cause trouble.
We will address that issue next.

Syntax Directed Left Rec

Syntax directed translation adds semantic rules to be carried out when syntactic rules are applied. Let's do conversion of infix to postfix.

Expr → Expr + Term	{out(" + ");}
| Term	
Term → Term * Factor	{out(" * ");}
| Factor	
Factor → (Expr)	
| int	{out(" ", int.val, " ");}

How It Works

Examples of applying previous syntax directed translation

Input: 15 + 20 + 7 * 3 + 2

Output: 15 20 + 7 3 * + 2 +

Input: 15 + 20 + 7 + 3 * 2

Output: 15 20 + 7 + 3 2 * +

Direct Placement of Actions

Expr → Term ExprRest

ExprRest → + Term ExprRest {out (“ + “);}

| ε

Term → Factor TermRest

TermRest → * Factor TermRest {out(“ * “);}

| ε

Factor → (Expr)

| int {out(“ “,int.val,” “);}

Problems Galore

Examples of applying previous syntax directed translation

Input: 15 + 20 + 7 * 3 + 2

Output: 15 20 7 3 * 2 + + +

Input: 15 + 20 + 7 + 3 * 2

Output: 15 20 7 3 2 * + + +

Treat Actions as Terminals

Expr → Term ExprRest

ExprRest → + Term {out (“ + “);} ExprRest

| ε

Term → Factor TermRest

TermRest → * Factor {out(“ * “);} TermRest

| ε

Factor → (Expr)

| int {out(“ “,int.val,” “);}

Top Down Parsing

Recursive Descent parsing uses recursive procedures to model the parse tree to be constructed. The parse tree is built from the top down, trying to construct a left-most derivation.

Beginning with ***start*** symbol, for each non-terminal (syntactic class) in the grammar a procedure which parses that syntactic class is constructed.

Consider the expression grammar (:

E → T E'

E' → **+** T E' | **ε**

T → F T'

T' → ***** F T' | **ε**

F → **(** E **)** | **id**

The following procedures have to be written:

Recursive Descent

```
Procedure E
  begin { E }
    call T
    call E'
end { E }

Procedure E'
  begin { E' }
    If token = "+" then
      begin { addition }
        nextsy
        call T
        out(" + ")
        call E'
      end { addition }
  end { E' }
```

```
Procedure T
  begin { T }
    call F
    call T'
  end { T }

Procedure T'
  begin { T' }
    If token = " * " then
      begin { multiply }
        nextsy()
        call F
        out(" * ")
        call T'
      end { multiply }
  end { T' }
```

```
Procedure F
  begin { F }
    case token is
    "(":
        nextsy()
        call E
        if token = ")" then
          nextsy()
        else
          ERROR()
    "id":
        out( id.val )
        Get next token
    otherwise:
        ERROR()
  end { F }
```

Process

- Write left recursive grammar with semantic actions.
- Rewrite a right recursive with actions treated as terminals in original rules.
- Develop recursive descent parser.

Left Factoring

When have rules like

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

which rule to choose is a problem

Factor as

$$A \rightarrow \alpha\, X$$

$$X \rightarrow \beta \mid \gamma$$

EBNF
(DAY #9,10)

EBNF

Extended Backus Naur Form (EBNF)

non-terminal ::= rhs

Where the rhs can include quoted terminals, non-terminal, designated keywords, and the special symbols

$s_1 \mid \ldots \mid s_k$	choose one of k strings
{ s }	repeat string s 0 or more times
[s]	optionally include string s

Pascal-S EBNF#1

RED indicates a reserved word or a special symbol

BLUE is an Identifier

Program ::= *program_heading*
block '.'

program_heading ::= **PROGRAM NAME** '(' *identifier_list* ')' ';'

identifier_list ::= **NAME** { ',' **NAME** }

block ::= *declaration_part statement_part*

declaration_part ::= [*constant_definition_part*]
[*type_definition_part*]
[*variable_declaration_part*]
procedure_and_function_declaration_part

Pascal-S EBNF#2

constant_definition_part ::= **CONST** *constant_definition* '**;**' { *constant_definition* '**;**' }
constant_definition ::= **NAME** '**=**' *constant*

constant ::= ['**+**' | '**-**'] (**CONSTANT_NAME** | **NUMBER**) | **STRING**

type_definition_part ::= **TYPE** *type_definition* '**;**' { *type_definition* '**;**' }
type_definition ::= **NAME** '**=**' *type*

variable_declaration_part ::= **VAR** *variable_declaration* '**;**' { *variable_declaration* '**;**' }
variable_declaration ::= *identifier_list* '**:**' *type*

procedure_and_function_declaration_part ::=
{ (*procedure_declaration* | *function_declaration*) '**;**' }

procedure_declaration ::= *procedure_heading* '**;**' *block*
function_declaration ::= *function_heading* '**;**' *block*

Pascal-S EBNF#3

type ::= *simple_type* | *structured_type* | **TYPE_NAME**

simple_type ::= *constant* '**..**' *constant*

structured_type ::= *array_type* | *record_type*

array_type ::= **ARRAY** '**[**' *index_type* { '**,**' *index_type* } '**]**' **OF** *element_type*
index_type ::= *simple_type*
element_type ::= *type*

record_type ::= **RECORD** *field_list* **END**
field_list ::= *record_section* { '**;**' *record_section* }
record_section ::= *identifier_list* '**:**' *type*

Pascal-S EBNF#4

procedure_heading ::= **PROCEDURE NAME** [*formal_parameter_list*]

function_heading ::= **FUNCTION NAME** [*formal_parameter_list*] '**:**' *result_type*

result_type ::= **TYPE_NAME**

formal_parameter_list ::= '**(**' *formal_parameter_section* { '**;**' *formal_parameter_section* } '**)**'
formal_parameter_section ::= [**VAR**]*identifier_list* '**:**' *parameter_type*
parameter_type ::= **TYPE_NAME**

Pascal-S EBNF#5

statement_part ::= **BEGIN** *statement_sequence* **END**
statement_sequence ::= *statement* { '**;**' *statement* }
statement ::= (*simple_statement* | *structured_statement*)
simple_statement ::= [*assignment_statement* | *procedure_statement*]

procedure_statement ::= **PROCEDURE_NAME** [*actual_parameter_list*]
actual_parameter_list ::= '**(**' *expression* { '**,**' *expression* } '**)**'

assignment_statement ::= (*variable_access* | **FUNCTION_NAME**) '**:=**' *expression*

variable_access ::= **ACCESS_NAME** { *end_access* }
end_access ::= { *array_access* | *record_access* | *function_parameters* }
array_access ::= '**[**' *expression_list* '**]**'
record_access ::= '**.**' *variable_access*
function_parameters ::= '**(**' [*expression_list*] '**)**'

expression_list ::= *expression* { '**,**' *expression* }

Pascal-S EBNF#6

expression ::= *simple_expression* [*relational_operator simple_expression*]
relational_operator ::= '**=**' | '**<>**' | '**<**' | '**<=**' | '**>**' | '**>=**'

simple_expression ::= ['**+**' | '**-**'] *term* { *addition_operator term* }
addition_operator ::= '**+**' | '**-**' | **OR**

term ::= *factor* { *multiplication_operator factor* }
multiplication_operator ::= '*****' | '**/**' | **DIV** | **MOD** | **AND**

factor ::= **NUMBER** | **STRING** | **CONSTANT_NAME**
| *variable_access* | *function_designator*
| '**(**' *expression* '**)**' | **NOT** *factor*

function_designator ::= **FUNCTION_NAME** [*actual_parameter_list*]

Pascal-S EBNF#7

structured_statement ::= *compound_statement* | *repetitive_statement* |
conditional_statement

compound_statement ::= **BEGIN** *statement_sequence* **END**

repetitive_statement ::= *while_statement* | *repeat_statement* | *for_statement*
while_statement ::= **WHILE** *expression* **DO** *statement*
repeat_statement ::= **REPEAT** *statement_sequence* **UNTIL** *expression*
for_statement ::= **FOR VARIABLE_NAME** ':=' *expression* (**TO** | **DOWNTO**) *expression*
DO *statement*

conditional_statement ::= *if_statement* | *case_statement*
if_statement ::= **IF** *expression* **THEN** *statement* [**ELSE** *statement*] .
case_statement ::= **CASE** *expression* **OF** *case_element* { ';' *case_element* } [';'] **END**
case_element ::= *case_label_list* ':' *statement*
case_label_list ::= *constant* { ',' *constant* }

SYNTAX GRAPHS (CHARTS)
RAILROAD CHARTS
(DAY #9,10)

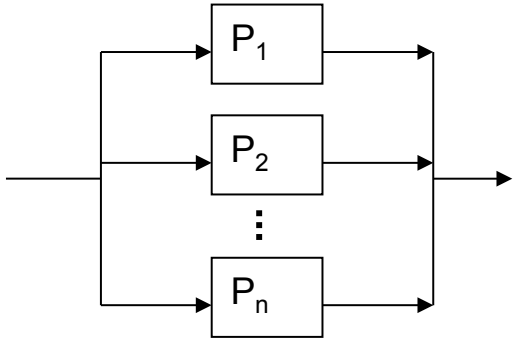
Syntax Graphs #1

Transforming a grammar expressed in EBNF to syntax graph (also called syntax chart or railroad chart) is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

Rules to construct a syntax graph:

R1.- Each non-terminal symbol A which can be expressed as a set of productions

A ::= P_1 | P_2 | . . . | P_n can be mapped into the following syntax graph:



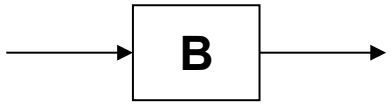
Syntax Graphs #2

Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

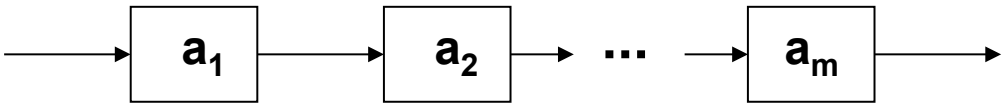
Rules to construct a syntax graph:

R2.- Every occurrence of a terminal symbol T in a P_i means that a token has been recognized and a new symbol (token) must be read. This is represented by a label T enclosed in a circle.

R3.- Every occurrence of a non-terminal symbol B in a P_i corresponds to an activation of the recognizer B.



R4.- A production P having the form $P = a_1\ a_2\ \ldots\ a_m$ can be represented by the graph:



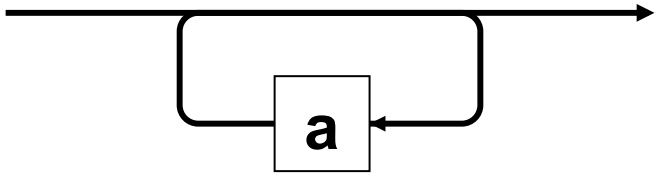
where every a_i is obtained by applying construction rules **R1** through **R6**

Syntax Graphs #3

Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

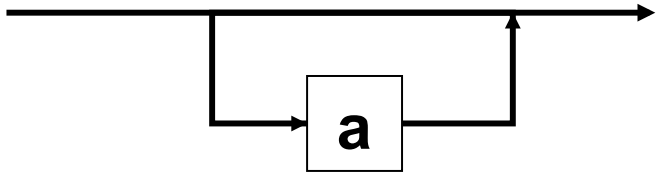
Rules to construct a syntax graph:

R5.- A production P having the form P = {a} can be represented by the graph:



where **a** is obtained by applying constructing rules R1 through R6

R6.- A production P having the form P = [a] can be represented by the graph:

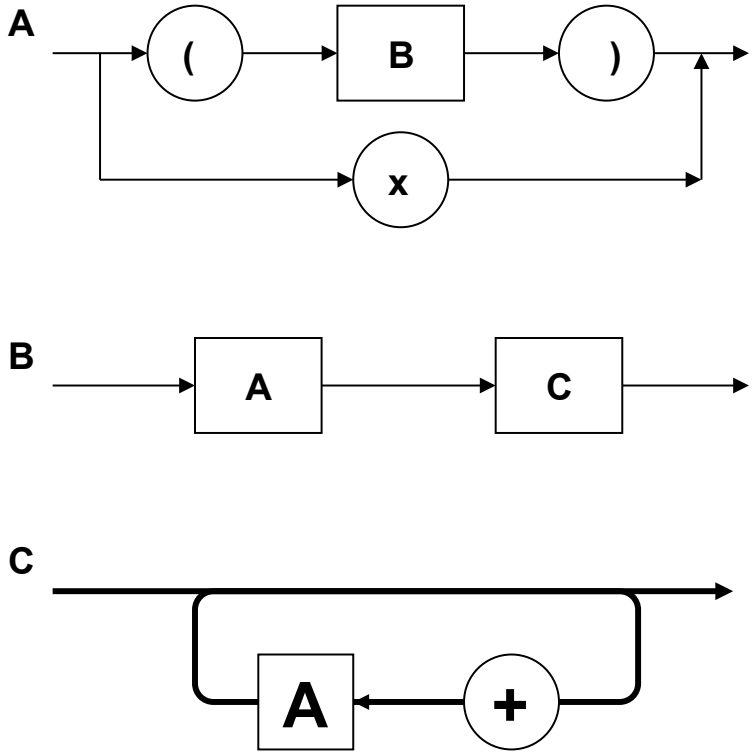


where **a** is obtained by applying constructing rules R1 through R6

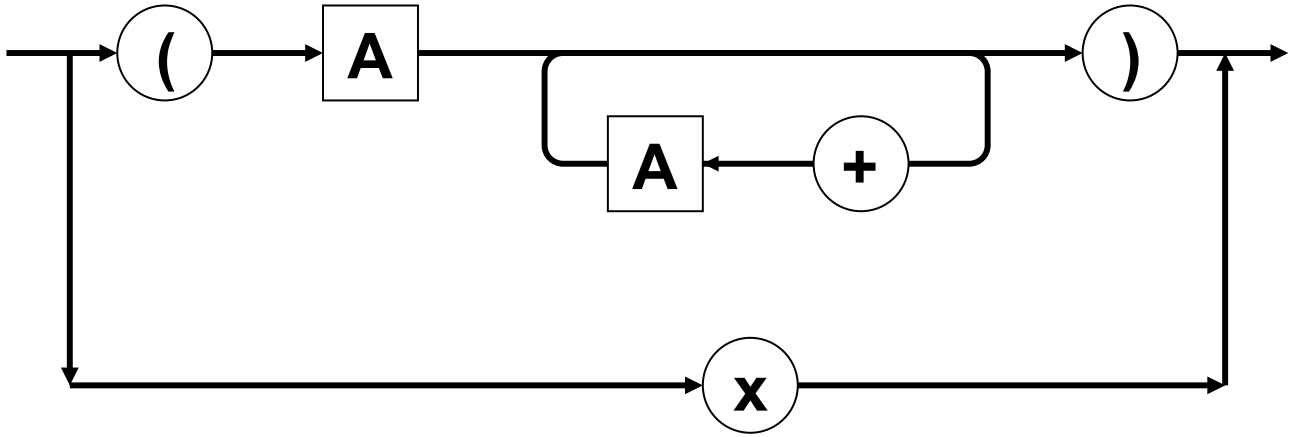
Syntax Graphs from EBNF

Example from N. Wirth:

A ::= “x” | “(“ B “)”
B ::= A C
C ::= { “+” A }



Strings from Syntax Graph



x
(x)
(x + x)

Parser from Syntax Graph#1

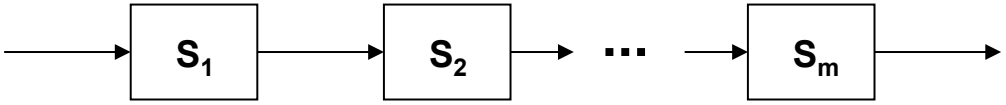
Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

Rules to construct a parser from a syntax graph (N. Wirth):

B1.- Reduce the system of graphs to as few individual graphs as possible by appropriate substitution.

B2.- Translate each graph into a procedure declaration according to the subsequent rules B3 through B7.

B3.- A sequence of elements



Is translated into the compound statement

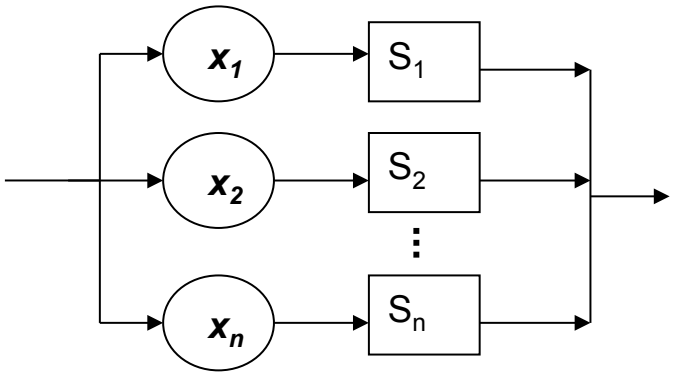
$$\{ T(S_1)\ T(S_2)\ \ldots\ T(S_n) \}$$

T(S) denotes the translation of graph S

Parser from Syntax Graph#2

Rules to construct a parser from a syntax graph:

B4.- A choice of elements



Conditional

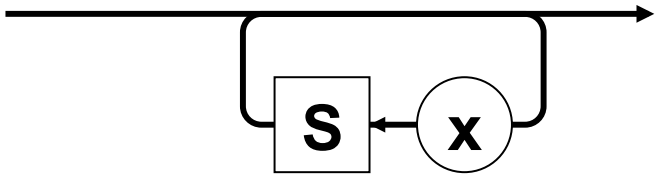
if sy in x_1 { insymbol(); T(S_1) } else
if sy in x_2 { insymbol(); T(S_2) } else
. . .

if sy in x_n { insymbol(); T(S_n) } else
***error*();**

Parser from Syntax Graph#3

Rules to construct a parser from a syntax graph:

B5.- A loop of the form



is translated into the statement

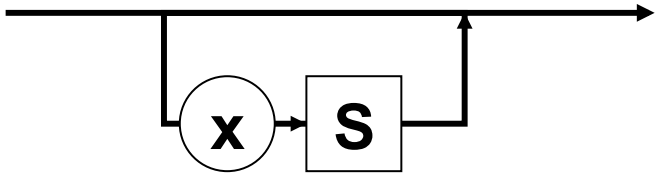
while sy in x do { insymbol(): T(S) }

where T(S) is the translation of S according to rules B3 through B7.

Parser from Syntax Graph#4

Rules to construct a parser from a syntax graph:

B6.- A loop of the form



is translated into the statement

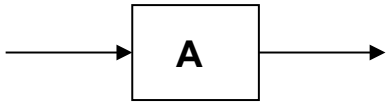
if x in L { insymbol(); T(S) }

where T(S) is the translation of S according to rules B3 through B8.

Parser from Syntax Graph#5

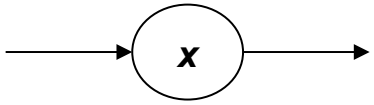
Rules to construct a parser from a syntax graph:

B7.- An element of the graph denoting another graph A



is translated into the procedure call statement A.

B8.- An element of the graph denoting a terminal symbol *x*



Is translated into the statement

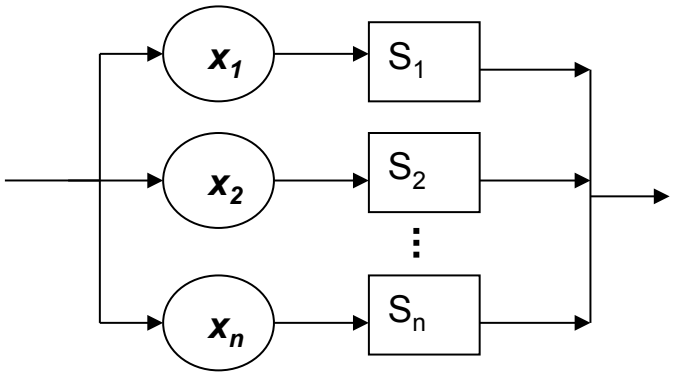
```
if (sy in x) insymbol(); else error();
```

Where error is a routine called when an ill-formed construct is encountered.

Parser from Syntax Graph#6

Useful variants of rules B4 and B5:

B4a.- A choice of elements



Conditional

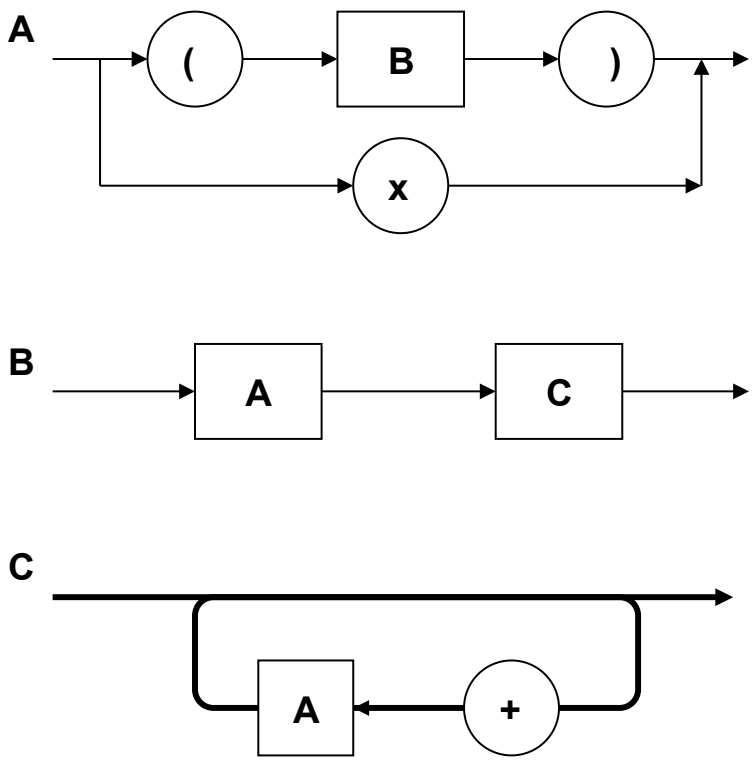
if sy in x_1 { insymbol(); T(S_1) } else
if sy in x_2 { insymbol(); T(S_2) } else
. . .

if sy in x_n { insymbol(); T(S_n) } else
***error*();**

Example Graphs

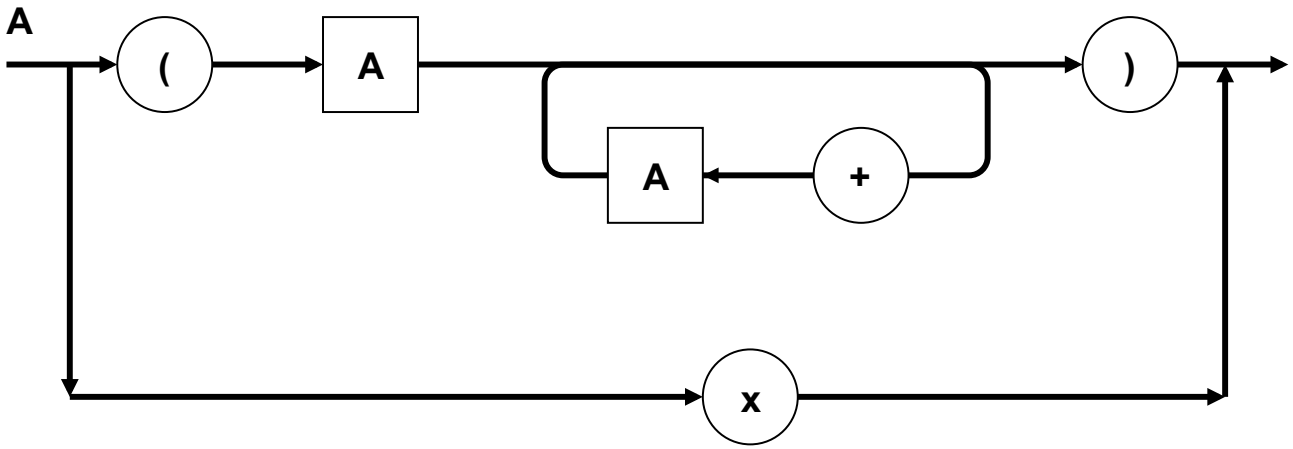
Applying the above mentioning rules to create one graph to this example:

A ::= “x” | “(“ B “)”
B ::= A C
C ::= { “+” A }



Combining Graphs

We will obtain this graph:



Using this graph and choosing from rules B1 to B8 a parser program can be generated.

Pseudo-code Parser

```
void function A() {
  if (sy == ‘x’) insymbol();  // ‘x’ is replaced by its token
  else if (sy == ‘(‘) {
    insymbol();
    A();
    while sy == ‘+’ {
      insymbol();
      A();
    }
    if (sy == ‘)’) insymbol(); else error(err_number);
  else error(err_number);
}
```

PASCAL-S RECURSIVE DESCENT (DAY #9,10)

Main for Pascal-S #1

```
(* First for some constructs *)
constbegsys := [plus, minus, intcon, realcon, charcon, ident];
typebegsys := [ident, arraysy, recordsy];
blockbegsys := [constsy, typesy, varsy, proceduresy, functionsy, beginsy];
facbegsys := [intcon, realcon, charcon, ident, lparent, notsy];
statbegsys := [beginsy, ifsy, whilesy, repeatsy, forsy, casesy];
stantyps := [notyp, ints, reals, bools, chars];
```

Main for Pascal-S #2

```
insymbol;
if sy <> programsy then error(3) else
  begin insymbol;
    if sy <> ident then error(2) else
    begin progname := id; insymbol;
      if sy <> lparent then error(9) else
      repeat insymbol;
        if sy <> ident then error(2) else insymbol
      until sy <> comma;
      if sy = rparent then insymbol else error(4);
    end
  end;
block(blockbegsys+statbegsys, false, 1);
if sy <> period then error(22);
emit(31); (* halt *)
```

Useful Tests

```
procedure skip(fsys: symset; n: integer);
begin error(n);
  while not (sy in fsys) do insymbol
end (* skip *);
procedure test(s1, s2: symset; n: integer);
begin
  if not (sy in s1) then skip(s1+s2, n)
end (* test *);
procedure testsemicolon;
begin
  if sy = semicolon then insymbol else
  begin error(14);
    if sy in [comma, colon] then insymbol
  end;
  test([ident]+blockbegsys, fsys, 6)
end (* testsemicolon *);
```

Part of Block

```
procedure block(fsys: symset; isfun: boolean; level: integer);
…..
 repeat
    if sy = constsy then constantdeclaration;
    if sy = typesy then typedeclaration;
    if sy = varsy then variabledeclaration;
    while sy in [proceduresy, functionsy] do procdeclaration;
    test([beginsy], blockbegsys+statbegsys, 56)
  until sy in statbegsys;
  insymbol; statement([semicolon, endsy]+fsys);
  while sy in [semicolon]+statbegsys do
    begin if sy = semicolon then insymbol else error(14);
      statement([semicolon, endsy]+fsys)
    end;
  if sy = endsy then insymbol else error(57);
  test(fsys+[period], [], 6)
end (* block *);
```

Statement#1

```
begin (* statement *)
  if sy in statbegsys+[ident] then
      case sy of
        ident:  begin i:= loc(id); insymbol;
                    if i <> 0 then
                    case tab[i].obj of
                      konstant, typel: error(45);
                      variable:
                          assignment(tab[i].lev, tab[i].adr);
                      prozedure:
                        if tab[i].lev <> 0 then call(fsys, i)
                              else standproc(tab[i].adr);
```

Statement#2

```
                funktion:
                  if tab[i].ref = display[level] then assignment(tab[i].lev+1, 0)
                  else error(45)
               end
             end;
      beginsy:   compoundstatement;
      ifsy:      ifstatement;
      casesy:    casestatement;
      whilesy:   whilestatement;
      repeatsy:  repeatstatement;
      forsy:     forstatement;
    end;
  test(fsys, [], 14)
end (* statement *)
```

Assignment Statement

```
procedure assignment(lv, ad: integer);
  var x,y: item; f: integer;
begin x.typ := tab[i].typ; x.ref := tab[i].ref;
  if tab[i].normal then f := 0 else f := 1;  emit2(f, lv, ad);
  if sy in [lbrack, lparent, period] then selector([becomes, egl]+fsys, x);
  if sy = becomes then insymbol else begin error(51); if sy = egl then insymbol end;
  expression(fsys, y);
  if x.typ = y.typ then
    if x.typ in stantyps then emit(38)
    else if x.ref <> y.ref then error(46)
    else if x.typ = arrays then emit1(23, atab[x.ref].size)
    else emit1(23, btab[x.ref].vsize)
  else if (x.typ=reals) and (y.typ=ints) then begin emit1(26, 0); emit(38) end
  else if (x.typ<>notyp) and (y.typ<>notyp) then error(46)
end (* assignment *);
```

Compound Statement

```
procedure compoundstatement;
begin insymbol;
  statement([semicolon, endsy]+fsys);
  while sy in [semicolon]+statbegsys do
  begin if sy = semicolon then insymbol else error(14);
    statement([semicolon, endsy]+fsys)
  end;
  if sy = endsy then insymbol else error(57)
end (* compoundstatement *);
```

IF Statement

```
procedure ifstatement;
  var x: item; lc1, lc2: integer;
begin insymbol;
  expression(fsys+[thensy, dosy], x);
  if not (x.typ in [bools, notyp]) then error(17);
  lc1 := lc; emit(11); (* jmpc *)
  if sy = thensy then insymbol else begin error(52); if sy = dosy then insymbol end;
  statement(fsys+[elsesy]);
  if sy = elsesy then
    begin insymbol; lc2 := lc; emit(10);
      code[lc1].y := lc; statement(fsys); code[lc2].y := lc
    end
  else code[lc1].y := lc
end (* if statment *);
```

While Statement

```
procedure whilestatement;
  var x: item; lc1, lc2: integer;
begin insymbol; lc1 := lc;
  expression(fsys+[dosy], x);
  if not (x.typ in [bools, notyp]) then error(17);
  lc2 := lc; emit(11);
  if sy = dosy then insymbol else error(54);
  statement(fsys); emit1(10, lc1); code[lc2].y := lc
end (* whilestatement *);
```

Repeat Statement

```
procedure repeatstatement;
  var x: item; lc1: integer;
begin lc1 := lc;
  insymbol; statement([semicolon, untilsy]+fsys);
  while sy in [semicolon]+statbegsys do
  begin if sy = semicolon then insymbol else error(14);
    statement([semicolon, untilsy]+fsys)
  end;
  if sy = untilsy then
    begin insymbol; expression(fsys, x);
      if not (x.typ in [bools, notyp]) then error(17);
      emit1(11, lc1)
    end
  else error(53)
end (* repeatstement *);
```

PREDICTIVE PARSING
FIRST AND FOLLOW SETS
(DAY #11)

First Set

A recursive descent (or predictive) parser chooses the correct production looking ahead at the input string a fixed number of symbols (typically one symbol or token).

First set:

Let α be a string of terminals and non-terminals.

First(α) is the set of all terminals that can begin strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is in First(α).

Example 1 of First

Example: Given the following expression grammar:

E → E + T | T

T → T * F | F

F → **(** E **)** | **id**

First(F) = { id, (}

First(T) = { id, (}

First(T * F) = { id, (}

First(T) = { id, (}

First(E + T) = { id, (}

Because: E + T → T + T → F + T → id + T

E + T → T + T → F + T → (E) + T

First(E) = { id, (}

Because: E → T → F → id

E → T → F → (E)

This creates a conflict on which to choose in a top down parser.

Nullables

Nullable symbols are the ones that produce the empty (ε) string

Example: Given the following grammar, find the nullable symbols and the First set:

Z → d	Y → ε	X → Y
Z → X Y Z	Y → c	X → a

Note that if X can derive the empty string, nullable(X) is true.

X → Y → ε

Y → ε		Nullable	First
Z → d	X	Yes	{a, c, ε}
Z → X Y Z	Y	Yes	{c, ε}
	Z	No	{a, c, d}

Computing First

- If X is a terminal, then First(X) = {X}
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, for k>=1, then place a in First(X) if for some i, a is in First(Y_i) and each of Y_1 through Y_{i-1} are nullable. If each of Y_1 through Y_k is nullable, then place ε in First(X).
- If $X \rightarrow \varepsilon$ is a production, add ε to First(X).

- Note: Some approaches use a separate status of nullable, rather than including ε in First.

Follow Set

Given a production **A**, Follow(**A**) is the set of terminal symbols that can immediately follow **A**. Note: **$** indicates end of input, so always in Follow of start symbol

Example: Given the following grammar:

Z → d	**Y → ε**	**X → Y**
Z → X Y Z	**Y → c**	**X → a**

Compute First, Follow, and nullable.

	Nullable	First	Follow
X	Yes	{ a, c, ε}	{ a, c, d }
Y	Yes	{c, ε}	{ a, c, d }
Z	No	{a, c, d }	{$ ←EOF}

Computing Follow

- Place $ in Follow(S), where S is start symbol.
- If there is some production A → αBβ, then everything in First(β) except ε is in Follow(B).
- If there is some production A → αB or a production A → αBβ, where ε is in First(β) then everything in Follow(A) is in Follow(B).

Expression Grammar

Example: Given the grammar:

E → E + T	T → T * F	F → **id**
E → T	T → F	F → **(** E **)**

We can rewrite the grammar to avoid left recursion obtaining:

E → T E'	T → F T'	F → **id**
E' → **+** T E'	T' → ***** F T'	F → **(** E **)**
E' → ε	T' → ε	

Compute First, Follow, and nullable.

	Nullable	First	Follow
E	No	{ **id** , **(** }	{ **)**, **$** }
E'	Yes	{ **+** , **ε** }	{ **)**, **$** }
T	No	{ **id** , **(** }	{ **)** , **+**, **$** }
T'	Yes	{ ***** , **ε** }	{ **)** , **+**, **$** }
F	No	{ **id** , **(** }	{ **)** , ***** , **+**, **$** }

Creating Parsing Table

- Input: Grammar G
- Output: Parsing table M
- For each production A → α, do
  - For each terminal a in First(α). Add A → α to M[A,a].
  - If ε is in First(α), then for each terminal b in Follow(A), add A → α to M[A,b]. Further if $ is in Follow(A), add A → α to M[A,$] as well.
  - All empty cells are errors.

Parse Table

Parsing table for the expression grammar:

	First	Follow			
E	{ id , (}	{), $ }	E → T E'	T → F T'	F → id
E'	{ + , ε }	{), $ }	E' → + T E'	T' → * F T'	F → (E)
T	{ id , (}	{) , +, $ }	E' → ε	T' → ε	
T'	{ * , ε }	{) , +, $ }			
F	{ id , (}	{) , * , +, $ }			

	+	*	id	(	)	$
E			E → T E'	E → T E'		
E'	E' → + T E'				E' → ε	E' → ε
T			T → F T'	T → F T'		
T'	T' → ε	T' → * F T'				T' → ε
F			F → id	F → (E)		

T' → ε

Table Entry to Code

Using the predictive parsing table, it is easy to write a recursive-descent parser:

	+	*	id	(	)	$
T'			T' → ε		T' → ε	T' → ε
		T' → * F T'				

```
void Tprime (void)
{
        switch  (token)
       {  case PLUS:       accept (PLUS) ; break ;
          case TIMES:      accept (TIMES) ; F ( )  ; Tprime ( );  break ;
          case RPAREN:     accept (RPAREN) ; break ;
          case EOF:        break ;
          default:         error ( ) ;
       }
}
```

Left Factoring (Again)

Avoid cases where two productions for same non-terminal start with same symbol.

Example: S → **if** E **then** S

S → **if** E **then** S **else** S

Solution: Left-factor the grammar. Take allowable ending "**else S**" and ε, and make a new production (new non-terminal) for them:

S → **if** E **then** S X

X → **else** S

X → e

Grammars whose predictive parsing tables contain no multiples entries are called LL(1).

The first L stands for left-to-right parse of input string. (input scanned from left to right)

The second L stands for leftmost derivation of the grammar. (apply production to leftmost non-terminal at each step of derivation)

The "1" stands for one symbol (token) lookahead

CKY (Cocke, Kasami, Younger)
$O(N^3)$ PARSING
(DAY #11)

Dynamic Programming

To solve a given problem, we solve small parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution.

The Parsing problem for arbitrary CFGs was elusive, in that its complexity was unknown until the late 1960s. In the meantime, theoreticians developed notion of simplified forms that were as powerful as arbitrary CFGs. The one most relevant here is the Chomsky Normal Form – CNF. It states that the only rule forms needed are:

A →	BC	where B and C are non-terminals
A →	a	where a is a terminal

This is provided the string of length zero is not part of the language.

CKY (Bottom-Up Technique)

Let the input string be a sequence of *n* letters a_1 ... a_n.
Let the grammar contain *r* terminal and nonterminal symbols R_1 ... R_r,
Let R_1 be the start symbol.
Let P[n,n,r] be an array of Booleans. Initialize all elements of P to false.
For each i = 1 to n
    For each unit production $R_j \rightarrow a_i$, set P[i,1,j] = true.
    For each i = 2 to n
        For each j = 1 to n-i+1
            For each k = 1 to i-1
                For each production R_A -> R_B R_C
                    If P[j,k,B] and P[j+k,i-k,C] then set P[j,i,A] = true
If P[1,n,1] is true then a_1 ... a_n is member of language
else a_1 ... a_n is not member of language

CKY Parser

Present the **CKY** recognition matrix for the string **abba** assuming the Chomsky Normal Form grammar, **G = ({S,A,B,C,D,E}, {a,b}, S, P)**, specified by the rules **P**:

S → AB | BA
A → CD | a
B → CE | b
C → a | b
D → AC
E → BC

	a	b	b	a
1	A,C	B,C	B,C	A,C
2	S,D	E	S,E	
3	B	B		
4	S,E			

Bottom Up vs Top Down

- Bottom-Up: Two stack operations
  - Shift (move input symbol to stack)
  - Reduce (replace top of stack α with A, when A$\rightarrow\alpha$)
  - Challenge is when to do shift or reduce and what reduce to do.
    - Can have both kinds of conflict
- Top-Down:
  - If top of stack is terminal
    - If same as input, read and pop
    - If not, we have an error
  - If top of stack is a non-terminal A
    - Replace A with some α, when A$\rightarrow\alpha$
    - Challenge is what A-rule to use

SIMPLE WHILE LANGUAGE
(DAY #13)

While.l #1

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <limits.h>

void setid(void);
void setnum(void);
%}
L                   [A-Za-z]
D                   [0-9]
ID                  {L}({L}|{D})*
NUM                 {D}+
```

While.l #2

```
%%
">="                    {ECHO; yylval = (int) '+'; return(RELOP);}
"<="                    {ECHO; yylval = (int) '-'; return(RELOP);}
"="|"#"|">"|"<"         {ECHO; yylval = (int) yytext[0]; return(RELOP);}
"+"                     {ECHO; return(PLUS);}
"-"                     {ECHO; return(MINUS);}
"*"                     {ECHO; return(TIMES);}
"/"                     {ECHO; return(DIVIDE);}
"{"                     {ECHO; return(LBRACE);}
"}"                     {ECHO; return(RBRACE);}
"("                     {ECHO; return(LPAREN);}
")"                     {ECHO; return(RPAREN);}
"["                     {ECHO; return(LBRACK);}
"]"                     {ECHO; return(RBRACK);}
```

While.l #3

```
";"             {ECHO; return(SEMICOLON);}
","             {ECHO; return(COMMA);}
":="            {ECHO; return(ASSIGN);}
"while"         {ECHO; return(WHILE);}
"do"            {ECHO; return(DO);}
"end"           {ECHO; return(END);}
"if"            {ECHO; return(IF);}
"then"          {ECHO; return(THEN);}
"else"          {ECHO; return(ELSE);}
{NUM}           {ECHO; setnum(); return(NUMBER);}
{ID}            {ECHO; setid(); return(IDENT);}
[ \t]*          {ECHO;}
[\n]+           {ECHO; dumpcode();}
.               {ECHO; printf("\nunrecognizable character\n");
                 return(BAD);}
```

While.l #4

```
%%
void setnum(void) {
  int i;
  yylval = 0; i = 0;
  while (yytext[i]) /* convert string to int */
    yylval = yylval*10 + ( (int) yytext[i++] - (int) '0' );
}
void setid(void) {
  char *p,*q,*r;
  p = idname; /* used to communicate string to syntax analyzer */
  q = yytext;   /* string found in input */
  r = symtab[0].name;         /* new symbol strats i zero-th slot */
  if (yyleng>=IDLENGTH)
    yytext[IDLENGTH-1] = '\0';                /* null termination of string makes copy safe */
  while (*r++ = ( *p++ = *q++ ));             /* copy new symbol into table */
  yylval = symsize;                           /* search bottom up -- small table so linear okay */
  while ( strcmp(symtab[--yylval].name, idname) ); /* always succeeds */
}
int yywrap(void) {
 return(1);
}
```

While.y #1

```
%token      SEMICOLON COMMA END LBRACE RBRACE LBRACK RBRACK
%token      RPAREN LPAREN NUMBER IDENT WHILE DO ASSIGN BAD
%token      IF THEN ELSE
%nonassoc   RELOP
%left       PLUS MINUS
%left       TIMES DIVIDE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <limits.h>
```

While.y #2

```
/* Constants */
#define IDLENGTH 9                  /* one more (for \0) than max ident length */
#define SYMMAX 20                   /* size of symbol table */
#define CODEMAX 100                 /* size of code table */

/* Code table */
struct {
        char opcode[IDLENGTH];      /* mnemonic of operator */
        int  first,second;          /* first and second operand */
} code[CODEMAX];                    /* code table */

/* Symbol table */
struct {
        char name[9];               /* identifier name */
        int  size;                  /* zero for scalar; number of elements for vector */
        int  countRead;             /* count the read references. */
        int  countWrite;            /* count the write references */
} symtab[SYMMAX];                   /* symbol table */
```

While.y #3

```
/* Other variables */
char        idname[IDLENGTH];        /* Ident just found -- first eight characters are
meaningful */
int         triple;                  /* Next available code slot */
int         showStart;                        /* start of code area yet to be
displayed */
int         symsize;                 /* next available symbol table slot */

int         forwardTest;        /* flag indicating cond jump forward or backward */
int         errorNo;                          /* error number counter */
int         warnNo;                           /* warning number counter */
```

While.y #4

```
/* Function declarations */
void basicinit(void);
void enter(char *name, int size);
void backpatch(int trip, int val);
void backpatch1(int trip, int val);
void emit(char *opcode, int first, int second);
void ok(int extra);
void fatal(char const *msg);
void dumpcode(void);
void countReport(void);
void error(char const *s);
void warning(char const *s);
void yyerror(char const *s);

%}
```

While.y #5

```
%%

program:
        startup decl body END
                                        { emit("EXIT",0,0); dumpcode();
                                          showStart = 1; dumpcode(); countReport();
                                        }
        ;
startup:
        /* empty */                     { basicinit(); }
        ;
decl:
        var.list SEMICOLON |
        SEMICOLON
        error                   { error("declaration syntax"); ok(COMMA); yyerrok; }
        ;
```

While.y #6

```
var.list:
        var.list COMMA var.def |
        var.def |
        error                           { error("declaration syntax"); ok(COMMA);
                                          yyerrok; }
        ;
var.def:
        IDENT LBRACK NUMBER RBRACK
                                        { if ($1 > 0) warning("duplicate name");
                                          else enter(idname,$3); } |
        IDENT
                                        { if ($1 > 0) warning("duplicate name");
                                          else enter(idname,0);
                                        }
        ;
```

While.y #7

```
body:
        statement.list
        ;
statement.list:
        statement.list statement |
        /* empty */
        ;
statement:
        assign SEMICOLON |
        while |
        do |
        if |
        LBRACE statement.list RBRACE |
        SEMICOLON |
        error                   { error("declaration syntax"); ok(LBRACE); yyerrok; }
```

While.y #8

```
while:
        while.start test DO statement
                                        { emit("JUMP",-$1,0);
                                          backpatch($2,-triple);
                                        }
        ;
while.start:
   WHILE                                { $$ = triple; forwardTest = 1; }
        ;
```

While.y #9

```
test:        exp RELOP exp      { emit("-",$1,$3);
                                  $$ = triple;
                                  switch( (char) $2) {
                                    case '=':    if (forwardTest) emit("JNZ",-triple+1,0);
                                                 else emit("JZ",-triple+1,0); break;
                                    case '#':     if (forwardTest) emit("JZ",-triple+1,0);
                                                 else emit("JNZ",-triple+1,0); break;
                                    case '>':    if (forwardTest) emit("JNP",-triple+1,0);
                                                 else emit("JP",-triple+1,0); break;
                                    case '<':    if (forwardTest) emit("JNM",-triple+1,0);
                                                 else emit("JM",-triple+1,0); break;
                                                    case '+':     if (forwardTest) emit("JM",-triple+1,0);
                                                 else emit("JNM",-triple+1,0); break;
                                    case '-':     if (forwardTest) emit("JP",-triple+1,0);
                                                 else emit("JNP",-triple+1,0); break;
                                   }
                                 }
                ;
```

While.y #10

```
do:
        do.start LBRACE statement.list RBRACE WHILE test
                                        { backpatch($6, $1); }
        ;
do.start:
        DO                              { $$ = triple; forwardTest = 0; }
        ;
if:
        IF {forwardTest=1;} test THEN statement {$$=triple;} optionalElse
                                        { backpatch($3, -$6); }
        ;
optionalElse:
        ELSE {$$=triple; emit("JUMP",0,0);} statement {backpatch1($2,-triple);} |
        /* empty */
        ;
```

While.y #11

```
assign:
        IDENT sub ASSIGN exp        { if ($1==0) {
                                        error("undefined"); emit("ERROR",0,0);
                                      }
                                      else if ($2==0) {
                                        if (symtab[$1].size>0) {
                                          error("need subscript"); emit("ERROR",0,0);
                                        }
                                        else emit(":=",$1,$4);
                                      }
                                      else if (symtab[$1].size==0) {
                                        error("cannot subscript"); emit("ERROR",0,0);
                                      }
                                      else {
                                        emit("[]=",$1,$2); emit("_",$4,0);
                                      }
                                    }
        ;
```

While.y #12

```
sub:
        LBRACK exp RBRACK
                                { $$ = $2; } |
        /* empty */
                                { $$ = 0; }
        ;
```

While.y #13

```
exp:        exp PLUS exp
                                    { $$ = -triple;
                                      emit("+",$1,$3);
                                    } |
            exp MINUS exp
                                    { $$ = -triple;
                                      emit("-",$1,$3);
                                    } |
            exp TIMES exp
                                    { $$ = -triple;
                                      emit("*",$1,$3);
                                    } |
            exp DIVIDE exp
                                    { $$ = -triple;
                                      emit("/",$1,$3);
                                    } |
            LPAREN exp RPAREN
                                    { $$ = $1; } |
```

While.y #14

```
NUMBER                { $$ = -triple; emit("con",$1,0); } |
IDENT sub             { $$ = -triple;
                        if ($1==0) {
                          error("undefined variable"); emit("ERROR",0,0);
                        }
                        else if ($2==0) {
                          if (symtab[$1].size>0) {
                            error("need subscript"); emit("ERROR",0,0);
                          }
                          else $$ = $1;
                        }
                        else if (symtab[$1].size==0) {
                           error("cannot subscript"); emit("ERROR",0,0);
                        }
                        else emit("=[]",$1,$2);
                      }
;
```

While.y #15

```
%%
void basicinit( void) {
  code[0].opcode[0] = '\0';
  showStart = 1;          /* Index of next code slot that has yet to be dumped */
  triple = 1;             /* Index of first available code slot */
  symsize = 1;                    /* Index of next available symbol table slot (zero-th reserved) */
  errorNo = 0;                    /* Number of errors detected */
  warnNo = 0;                     /* Number of warnings given */
}

void enter(char *name, int size) {
  char *p;
  if (symsize<SYMMAX) { symtab[symsize].size = size;
    symtab[symsize].countRead  = 0; symtab[symsize].countWrite = 0;
    p = symtab[symsize++].name; while (*p++ = *name++);
  }
  else fatal("symbol table too large");
}
```

While.y #16

```
void backpatch(int trip, int val) {
  code[trip].second = val;
}

void backpatch1(int trip, int val) { /* for backpatch the first field */
  code[trip].first = val;
}
```

While.y #17

```
void emit(char *opcode, int first, int second) { /* generates code and handles var count */
  char *p;
  if (triple<CODEMAX) {
     if ( strcmp(opcode, "con") ) {          /* not a constant, so need to do further checks */
            if (second>0)                    /* 2nd var only used for read */
                symtab[second].countRead++; /* rvalue; note: a[a1]:=b; a1 is read */
       if ( !strcmp(opcode, ":=") || !strcmp(opcode, "[]=") )
                symtab[first].countWrite++;          /* lvalue */
       else if (first>0)
                symtab[first].countRead++;           /* rvalue; first operand  */
     }
     code[triple].first = first; code[triple].second = second;
     p = code[triple++].opcode; while (*p++ = *opcode++);
    }
   else fatal("code table is too large\n");
}
```

While.y #18

```
void emit(char *opcode, int first, int second) { /* generates code and handles var count */
  char *p;
  if (triple<CODEMAX) {
     if ( strcmp(opcode, "con") ) {          /* not a constant, so need to do further checks */
           if (second>0)                     /* 2nd var only used for read */
              symtab[second].countRead++; /* rvalue; note: a[a1]:=b; a1 is read */
      if ( !strcmp(opcode, ":=") || !strcmp(opcode, "[]=") )
              symtab[first].countWrite++;          /* lvalue */
      else if (first>0)
              symtab[first].countRead++;           /* rvalue; first operand  */
     }
     code[triple].first = first; code[triple].second = second;
     p = code[triple++].opcode; while (*p++ = *opcode++);
    }
   else fatal("code table is too large\n");
}
```

While.y #19

```
void ok(int extra) {
   while ((yychar!=WHILE) && (yychar!=SEMICOLON) && (yychar!=extra) &&
                (yychar>0)) yychar = yylex();
}
void fatal(char const *msg) {
  printf("%s\n",msg);
  dumpcode();
}
void dumpcode(void) {
  int i;
  printf("\n");
  for (i=showStart;i<triple;i++)
    printf("%4d : %8s %4d %4d\n",i,code[i].opcode,code[i].first,code[i].second);
  showStart = triple;
}
```

While.y #20

```
void countReport(void) {
  int i;
  printf("\n*** NUMBER OF ERRORS FOUND: %d *** \n", errorNo);
  printf(  "*** NUMBER OF WARNINGS: %d *** \n", warnNo);
  /* dump out symbol table with reference counts */
  for (i=1; i<symsize; i++)
              if (symtab[i].size>0)
                printf("%s[%d], # of read ref = %d, # of write ref = %d\n",
                           symtab[i].name, symtab[i].size, symtab[i].countRead,
                           symtab[i].countWrite);
              else
                printf("%s, # of read ref = %d, # of write ref = %d\n",
                           symtab[i].name, symtab[i].countRead, symtab[i].countWrite);
}
```

While.y #21

```
void error(char const *s) {
  printf("\nError: %s\n", s);
  errorNo++;
}

void warning(char const *s) {
  printf("\nWarning: %s\n", s);
  warnNo++;
}

void yyerror(char const *s) {
  printf("%s\n", s);
}
```

While.y #22

```
#include "while.lex.c"

int main(int argc, char **argv ) {
  int result;
  ++argv, --argc; /* skip over program name */
  if ( argc > 0 ) yyin = fopen( argv[0], "r" );
  else yyin = stdin;
  result = yyparse();
  system("pause()");
  return(result);
}
```

While.y #23

```
var1,var2,a[5],b[7];
var1 := var2+10; var1 := 9;
  1 :     con   10    0
  2 :      +    2   -1
  3 :     :=    1   -2
  4 :     con    9    0
  5 :     :=    1   -4
var1 := 1;
  6 :     con    1    0
  7 :     :=    1   -6
while var1<5 do
  8 :     con    5    0
  9 :      -    1   -8
 10 :     JNM   -9    0
{ a[var1] := var1*var1 + 1;
 11 :      *    1    1
 12 :     con    1    0
 13 :      +  -11  -12
 14 :     []=    3    1
 15 :      _  -13    0
 b[var1] := a[var1] + var2;
 16 :     =[]    3    1
 17 :      +  -16    2
 18 :     []=    4    1
 19 :      _  -17    0
 var1 := var1+1;
 20 :     con    1    0
 21 :      +    1  -20
 22 :     :=    1  -21
}
 23 :    JUMP   -8    0
end
 24 :    EXIT    0    0
```

While.y #23

*** NUMBER OF ERRORS FOUND: 0 ***

*** NUMBER OF WARNINGS: 0 ***

var1, # of read ref = 7, # of write ref = 4

var2, # of read ref = 2, # of write ref = 0

a[5], # of read ref = 1, # of write ref = 1

b[7], # of read ref = 0, # of write ref = 1

Bottom-Up Parsing

Days#16, 17

Reductions

- Top-down focuses on producing an input string from the start symbol
- Bottom-up focuses on reducing the string to the start symbol
- By definition, reduction is the reverse of production

Handle Pruning

- Bottom-up reverses a rightmost derivation since rightmost rewrites the leftmost non-terminal last
- Bottom-up must identify a **handle** of a sentential form (a string of terminals and non-terminals derived from the start symbol), where the handle is the substring that was replaced at the last step in a rightmost derivation leading to this sentential form.
- A handle must match the body (rhs) of some production
- Formally, if $S \Rightarrow_{rm}^{*} \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$ where $A \rightarrow \beta$ then β, in the position following α, is a handle of $\alpha\beta\omega$
- We would like handles to be unique, and they are so in unambiguous grammars
- **Handle pruning** is the process of reducing a sentential form to a deriving sentential form by reversing the last production

shift/reduce Parsing

- This involves a stack that holds the left part of a sentential for with the input holding the right part
- Initially the stack has a bottom of stack marker and the input is the entire string to be parsed, plus an end marker
  Stack = $ Input = w$
- Our goal is to consume the string and end up with the start symbol on stack
  Stack = $S Input = $

shift/reduce Process

- The process is one where we can either
  - Shift the next input symbol onto stack
  - Reduce “handle” on top of stack
  - Accept if successfully get to start symbol with all input consumed
  - Error is a syntax error is discovered

Conflicts in shift/reduce

- Handle pruning can encounter two types of conflicts
  - **reduce/reduce** is when there are two possible reductions and we cannot decide which to use
  - **shift/reduce** conflict is when we cannot decided whether to shift or reduce

Classic shift/reduce

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| **other**

Stack = **$… if** *expr* **then** *stmt*

Input = **else … $**

Should we shift **else** into stack or reduce??

Can prefer shift over reduce, but that may not work as a general policy

Classic reduce/reduce

If have two types of expression lists preceded by an id. One is an array reference and the other is a function call. Both can appear by themselves in C.

Relevant rules are:

stmt	→	**id (** *p_list* **)**
	|	*expr*
p_list	→	*p_list parm* | *parm*
e_list	→	*e_list parm* | *expr*
expr	→	**id (** *e_list*) **| id**
parm	→	**id**

Stack = **$...id(id** Input = **, id)…$**

Is this first *expr* or a *parm*?

One solution is that we differentiate **procid** from **id** in symbol table and hence via lexical analysis. Then the third symbol in stack, not part of handle, determines the reduction. The key is context.

Our Goal

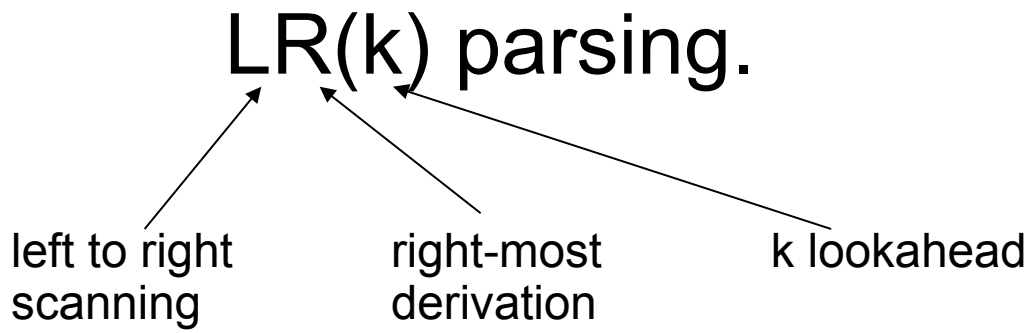
Find a useful subset of context free grammars that

1. Covers all or at least most unambiguous CF languages
2. Is easy to recognize
3. Avoids conflicts without severely limiting expressiveness
4. Is amenable to a fast parsing algorithm

LR Parsing

Days#18,19

LR Parsing

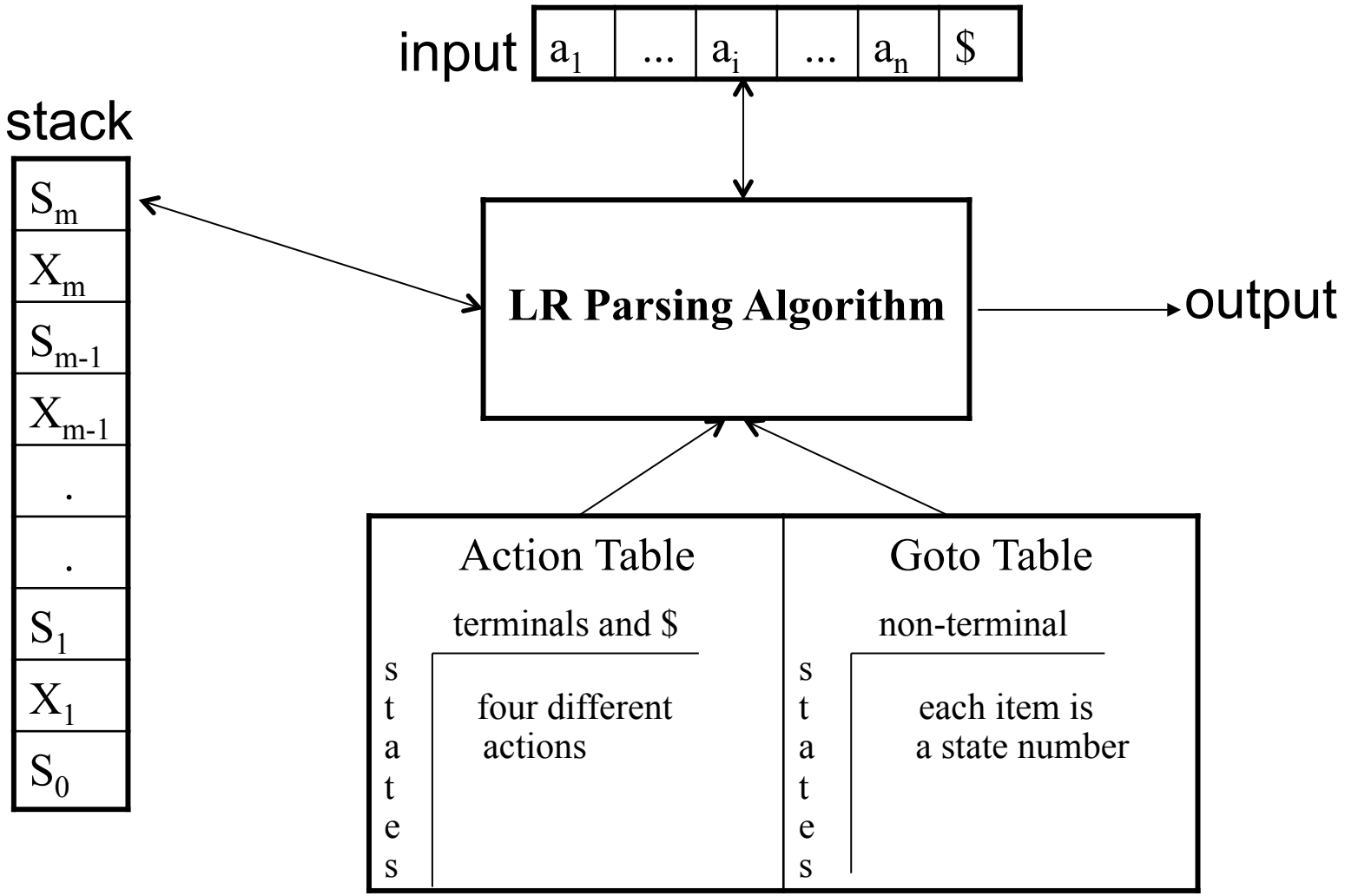


- LR is associated with bottom-up; LL with top-down
- LL(k), k>1, languages ⊃ LL(k-1) languages
- LR(1) languages ⊃ LL(k) languages, k ≥ 0
- LR(k), k>1, languages = LR(1) languages
- However, LR(k), k>1, grammars ⊃ LR(k-1) grammars
- LR grammars can find errors quickly, but they do not always have good context to recover

LR Parser Types

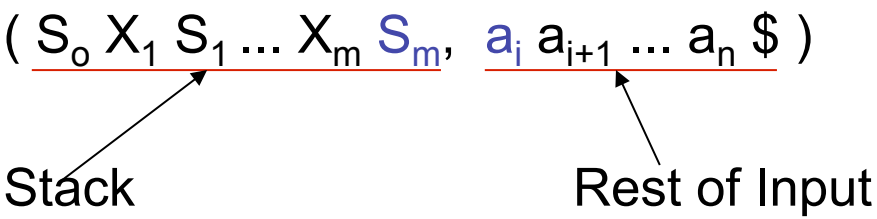
- SLR – simple LR parser
- LALR –look-head LR parser
- LR – most general LR parser
- SLR, LALR and LR are closely related
  - The parsing algorithm is the same
  - Their parsing tables are different

LR Parsing Algorithm



Configuration of LR Algorithm

- A configuration of a LR parsing is:



- S_m and a_i decide the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

X_1 ... X_m a_i a_{i+1} ... a_n $

Actions of LR-Parser

1. **shift s** -- shifts the next input symbol onto the stack. Shift is performed only if **action[s_m,a_i] = sk**, where k is the new state. In this case

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_m S_m a_i k, a_{i+1} ... a_n \$)

2. **reduce A→β** (if **action[s_m,a_i] = rn** where n is a production number)
   - pop 2|β| items from the stack;
   - then push **A** and **k** where **k=goto[$s_{m-|\beta|}$,A]**

(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... $X_{m-|\beta|}$ $S_{m-|\beta|}$ A k, a_i ... a_n \$)

   - Output is the reducing production reduce A→β or the associated semantic action or both

3. **Accept** – Parsing successfully completed
4. **Error** -- Parser detected an error (empty entry in action table)

Reduce Action

- pop $2|\beta|$ (=j) items from the stack; let us assume that $\beta=Y_1Y_2...Y_j$
- then push **A** and **s** where **$s=goto[s_{m-j},A]$**

  $(S_o X_1 S_1 ... X_{m-j} S_{m-j} Y_1 S_{m-j+1} ...Y_j S_m, a_i a_{i+1} ... a_n \$)$

  ➔ $(S_o X_1 S_1 ... X_{m-j} S_{m-j} A s, a_i ... a_n \$)$

- In fact, $Y_1Y_2...Y_j$ is a handle.

  $X_1 ... X_{m-j} A a_i ... a_n \$ \Rightarrow X_1 ... X_{m-j} Y_1...Y_j a_i a_{i+1} ... a_n \$$

Expression Grammar

Example: Given the grammar:

E → E + T	T → T * F	F → **id**
E → T	T → F	F → **(** E **)**

Compute Follow.

	Follow
E	{ **)**, **+**, **$** }
T	{ **)** , *, **+**, **$** }
F	{ **)** , * , **+**, **$** }

SLR Parsing Tables

- An **LR(0) item** of a grammar G is a production of G with a dot at some position of the right side.
- Ex: $A \rightarrow aBb$ LR(0) Items: $A \rightarrow \cdot aBb$, $A \rightarrow a \cdot Bb$, $A \rightarrow aB \cdot b$, $A \rightarrow aBb \cdot$
- Sets of LR(0) items will be the states of action and goto tables of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from ***I*** by the two rules:

  1. Initially, every LR(0) item in ***I*** is added to ***closure(I)***.
  2. If $\mathbf{A} \rightarrow \alpha \blacksquare \mathbf{B}\beta$ is in ***closure(I)*** and $\mathbf{B} \rightarrow \gamma$ is a production rule of G; then $\mathbf{B} \rightarrow \blacksquare \gamma$ will be in the ***closure(I)***. We will apply this rule until no more new LR(0) items can be added to ***closure(I)***.

Closure Example

E’ → E
E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

closure({E’ → .E}) =
{ E’ → .E kernel item
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id }

Closure Algorithm

```
function closure ( I )
begin
        J := I;
        repeat
                for each item A → α.Bβ in J and each production
                        B→γ  of G such that B→.γ is not in J do
                                add B→.γ  to J;
        until no more items can be added to J;
        return J;
end
```

Goto Function

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

If **A** → α ■ **X**β in I then every item in **closure({A** → α**X** ■ β**})** will be in goto(I,X).

If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

Example:

I ={ E' → ■ E, E → ■ E+T, E → ■ T,
T → ■ T*F, T → ■ F, F → ■ (E), F → ■ id }

goto(I,E) = { E' → E ■ , E → E ■ +T }

goto(I,T) = { E → T ■ , T → T ■ *F }

goto(I,F) = {T → F ■ }

goto(I,() = { F → (■ E), E → ■ E+T, E → ■ T, T → ■ T*F, T → ■ F,
F → ■ (E), F → ■ id }

goto(I,id) = { F → id ■ }

Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- ***Algorithm***:

  C is { closure({S'→ ■ S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***
- The goto function is a deterministic FSA (finite state automaton), DFA, on the sets in C.

Canonical LR(0) Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

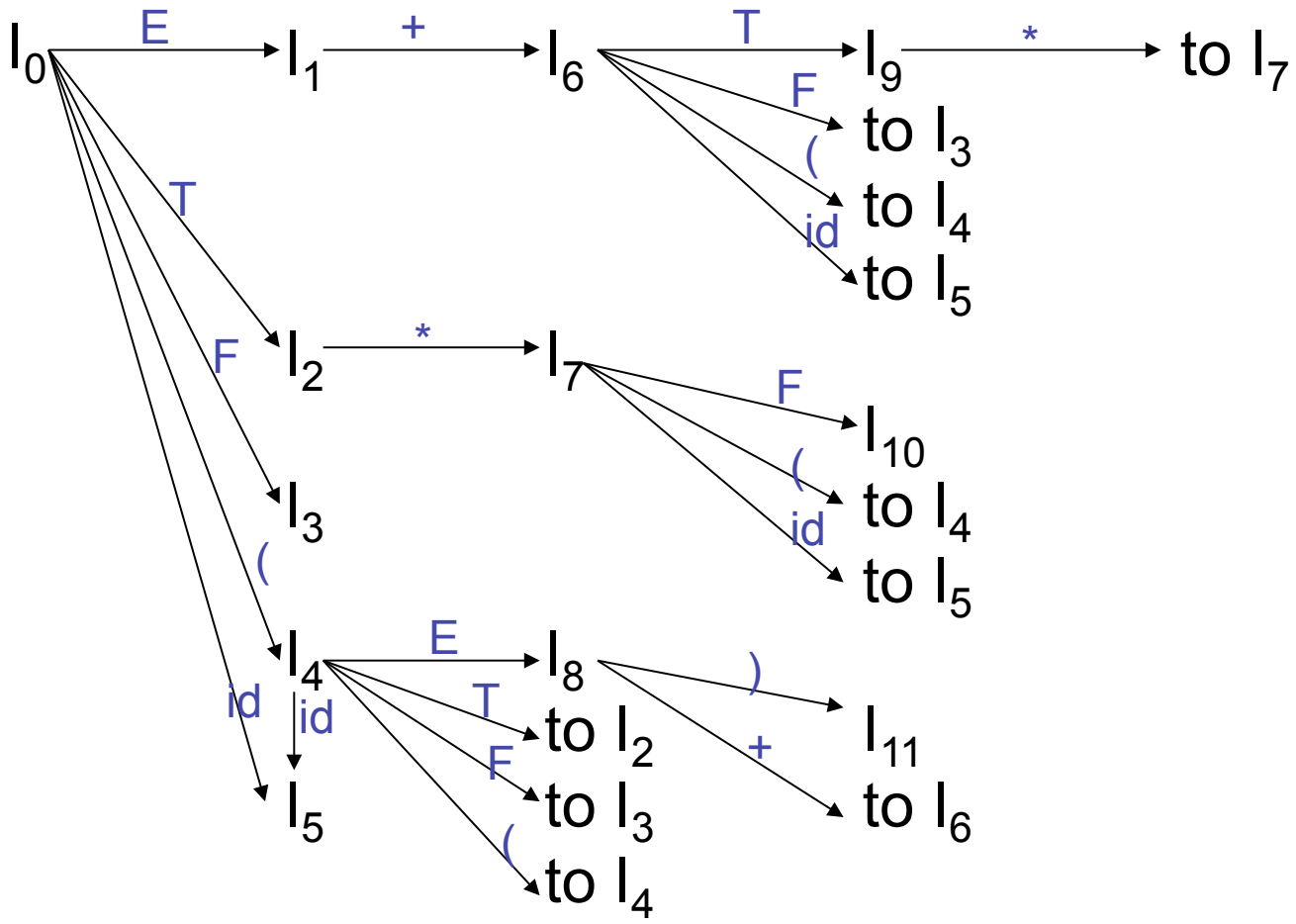
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

DFA of Goto Function



Compute SLR Parsing Table

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,\ldots,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, **$A \rightarrow \alpha.a\beta$** in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If **$A \rightarrow \alpha.$** is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in **FOLLOW(A)** where **$A \neq S'$**.
   - If **$S' \rightarrow S.$** is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S$

(SLR) Parsing Tables

0) E' → E
1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

Action Table | Goto Table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Actions of SLR-Parser

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called an SLR(1) grammar.
- Every SLR grammar is unambiguous, but every unambiguous grammar is not an SLR grammar.

Conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example 1

S → L=R
S → R
L→ *R
L → id
R → L

I_0: S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1: S' → S.

I_2: S → L.=R
R → L.

I_3: S → R.

I_4: L → *.R
R → .L
L → .*R
L → .id

I_5: L → id.

I_6: S → L=.R
R → .L
L→ .*R
L → .id

I_7: L → *R.

I_8: R → L.

I_9: S → L=R.

Problem

FOLLOW(R)={=,$}

= → shift 6

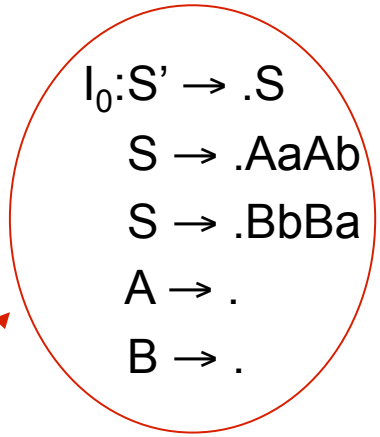
= → reduce by

shift/reduce conflict

Action[2,=] = shift 6
Action[2,=] = reduce by R → L
[S ⇒L=R ⇒*R=R] so follow(R) contains =

Conflict Example2

S → AaAb
S → BbBa
A → ε
B → ε



Problem

FOLLOW(A)={a,b}
FOLLOW(B)={a,b}

a → reduce by A → ε
a → reduce by B → ε
reduce/reduce conflict

b → reduce by A → ε
b → reduce by B → ε
reduce/reduce conflict

SLR Weakness

- In SLR method, state i makes a reduction by $A \rightarrow \alpha$ when the current token is **a**:
  - if $A \rightarrow \alpha.$ is in I_i and **a** is in FOLLOW(A)
- In some situations, βA cannot be followed by the terminal **a** in a right-sentential form when $\beta\alpha$ and the state i are on the stack top. This means that making reduction in this case is not correct.

LR(1) Item

- To avoid some invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

  $A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item (**a** is a terminal or end-marker.)
- Such an object is called an LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect on an item of the form $[A \rightarrow \alpha.\beta,a]$, where β is not $\in$.
  - But an item of the form $[A \rightarrow \alpha.,a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.
  - The set of such a's will be a subset of FOLLOW(A), and could be proper.

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A) as with SLR).
- A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq FOLLOW(A)$

  ...

  $A \rightarrow \alpha.,a_n$

Canonical Collection

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha . B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a rule of G; then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

    for each I in ***C*** and each grammar symbol X

      if goto(I,X) is not empty and not in ***C***

        add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

Short Notation

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha.\beta,a_1$$

...

$$A \rightarrow \alpha.\beta,a_n$$

can be written as

$$A \rightarrow \alpha.\beta,a_1/a_2/.../a_n$$

Canonical LR(1) Collection

S → AaAb
S → BbBa
A → ε
B → ε

I_0: S' → .S ,$
S → .AaAb ,$
S → .BbBa ,$
A → . ,a
B → . ,b

S → I_1: S' → S. ,$
A → I_2: S → A.aAb ,$ —a→ to I_4
B → I_3: S → B.bBa ,$ —b→ to I_5

I_4: S → Aa.Ab ,$
A → . ,b
—A→ I_6: S → AaA.b ,$ —b→ I_8: S → AaAb. ,$

I_5: S → Bb.Ba ,$
B → . ,a
—B→ I_7: S → BbB.a ,$ —a→ I_9: S → BbBa. ,$

An Example

1. S' → S
2. S → C C
3. C → c C
4. C → d

I_0: closure({(S' → • S, $)}) =
(S' → • S, $)
(S → • C C, $)
(C → • c C, c/d)
(C → • d, c/d)

I_1: goto(I_0, S) = (S' → S • , $)

I_2: goto(I_0, C) =
(S → C • C, $)
(C → • c C, $)
(C → • d, $)

I_3: goto(I_0, c) =
(C → c • C, c/d)
(C → • c C, c/d)
(C → • d, c/d)

I_4: goto(I_0, d) =
(C → d •, c/d)

I_5: goto(I_2, C) =
(S → C C •, $)

An Example

I_6: goto(I_3, c) =
(C → c • C, $)
(C → • c C, $)
(C → • d, $)

I_7: goto(I_3, d) =
(C → d •, $)

I_8: goto(I_4, C) =
(C → c C •, c/d)

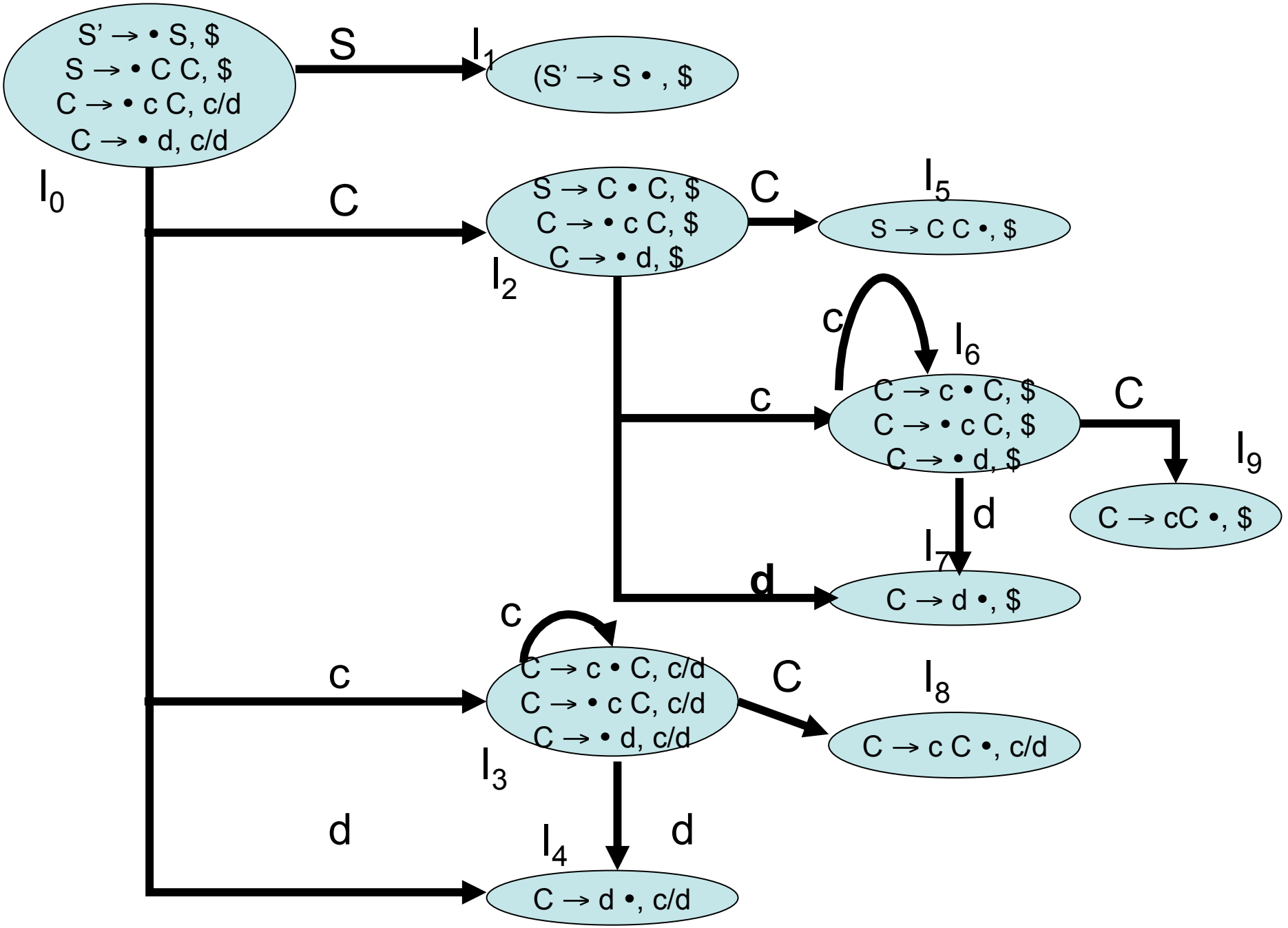
: goto(I_4, c) = I_4

: goto(I_4, d) = I_5

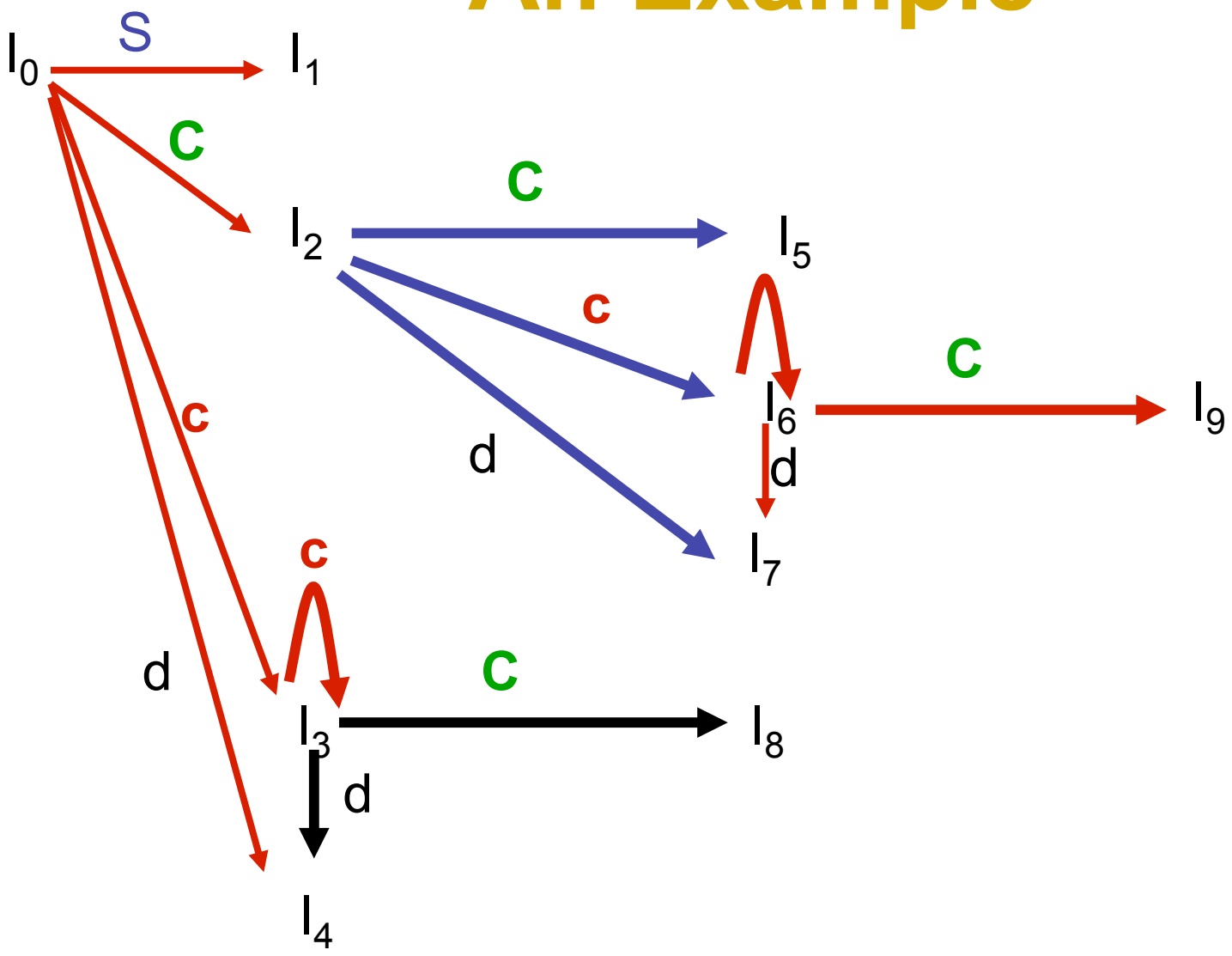
I_9: goto(I_7, c) =
(C → c C •, $)

: goto(I_7, c) = I_7

: goto(I_7, d) = I_8


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
C → d •, $
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I8
C → c C •, c/d
d
I4
d
C → d •, c/d

An Example



An Example

	c	d	$	S	C
0	s3	s4		1	2
1			a		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

The Core of LR(1) Items

- The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

  { (C → c • C, c/d),
  (C → • c C, c/d),
  (C → • d, c/d) }

  is { C → c • C,
  C → • c C,
  C → • d }

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'. $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \blacksquare a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha \blacksquare,a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S \blacksquare,\$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions are generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S,\$$

LALR Parsing Tables

1. **LALR** stands for **Lookahead LR.**

2. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.

3. The number of states in SLR and LALR parsing tables for a grammar G are equal.

4. But LALR parsers recognize more grammars than SLR parsers.

5. ***Bison*** creates a LALR parser for the given grammar.

6. A state of an LALR parser will again be a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \blacksquare = R, \$$ ➔ $S \rightarrow L \blacksquare = R$ ← Core

$R \rightarrow L \blacksquare , \$$ $R \rightarrow L \blacksquare$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id \blacksquare , =$ A new state: I_{12}: $L \rightarrow id \blacksquare , =$

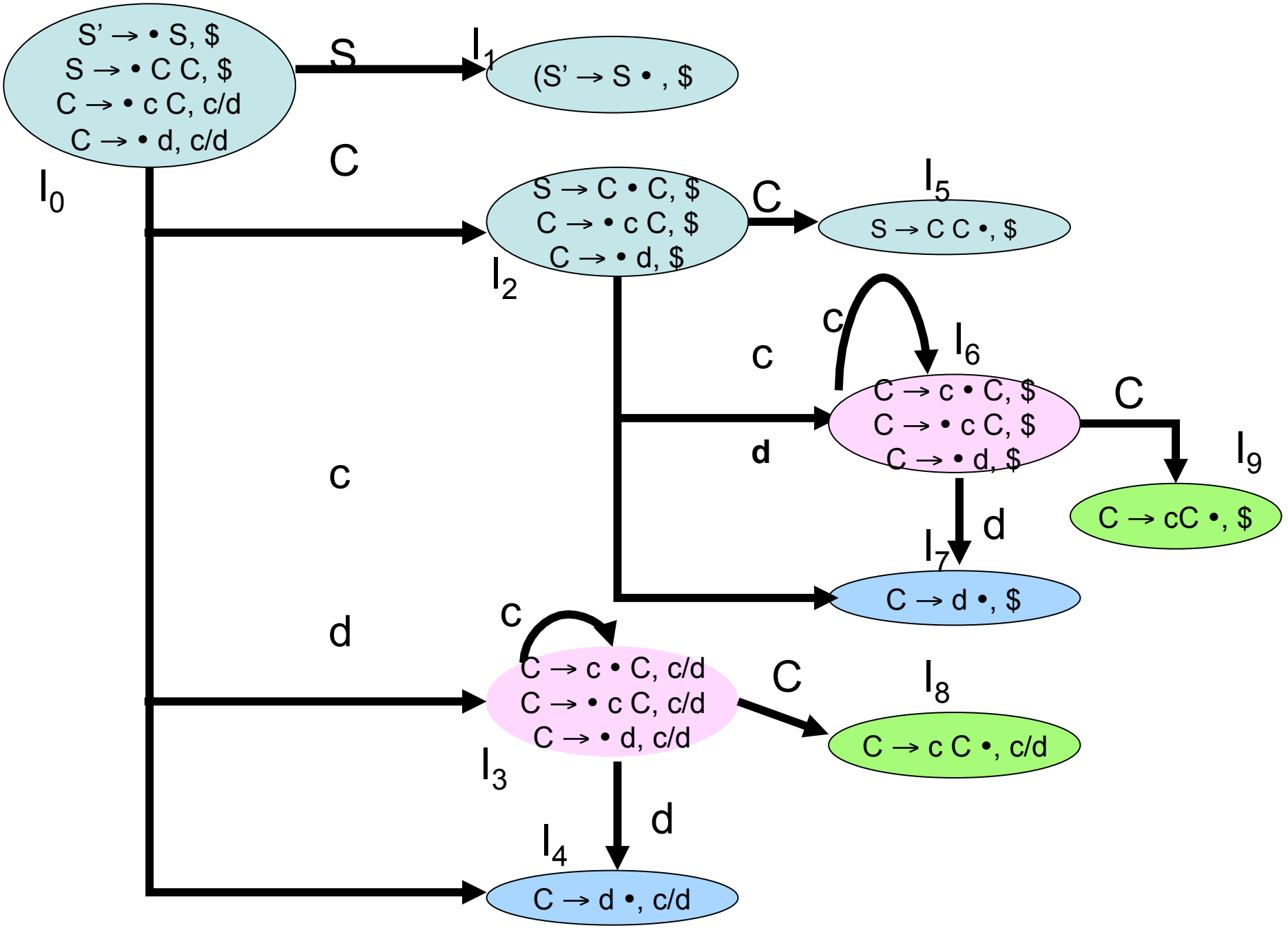
➔ $L \rightarrow id \blacksquare , \$$

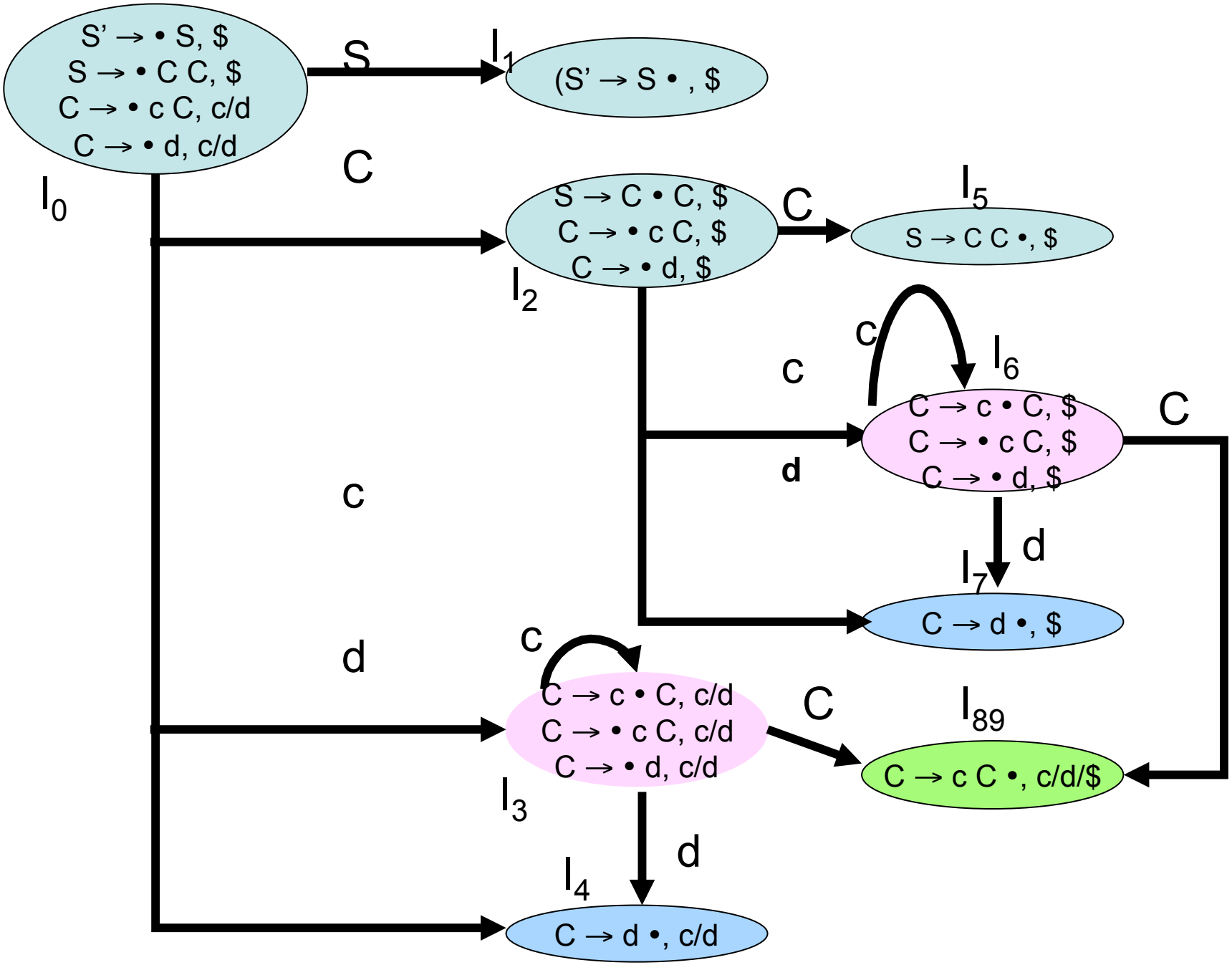
I_2: $L \rightarrow id \blacksquare , \$$ have same core, merge them

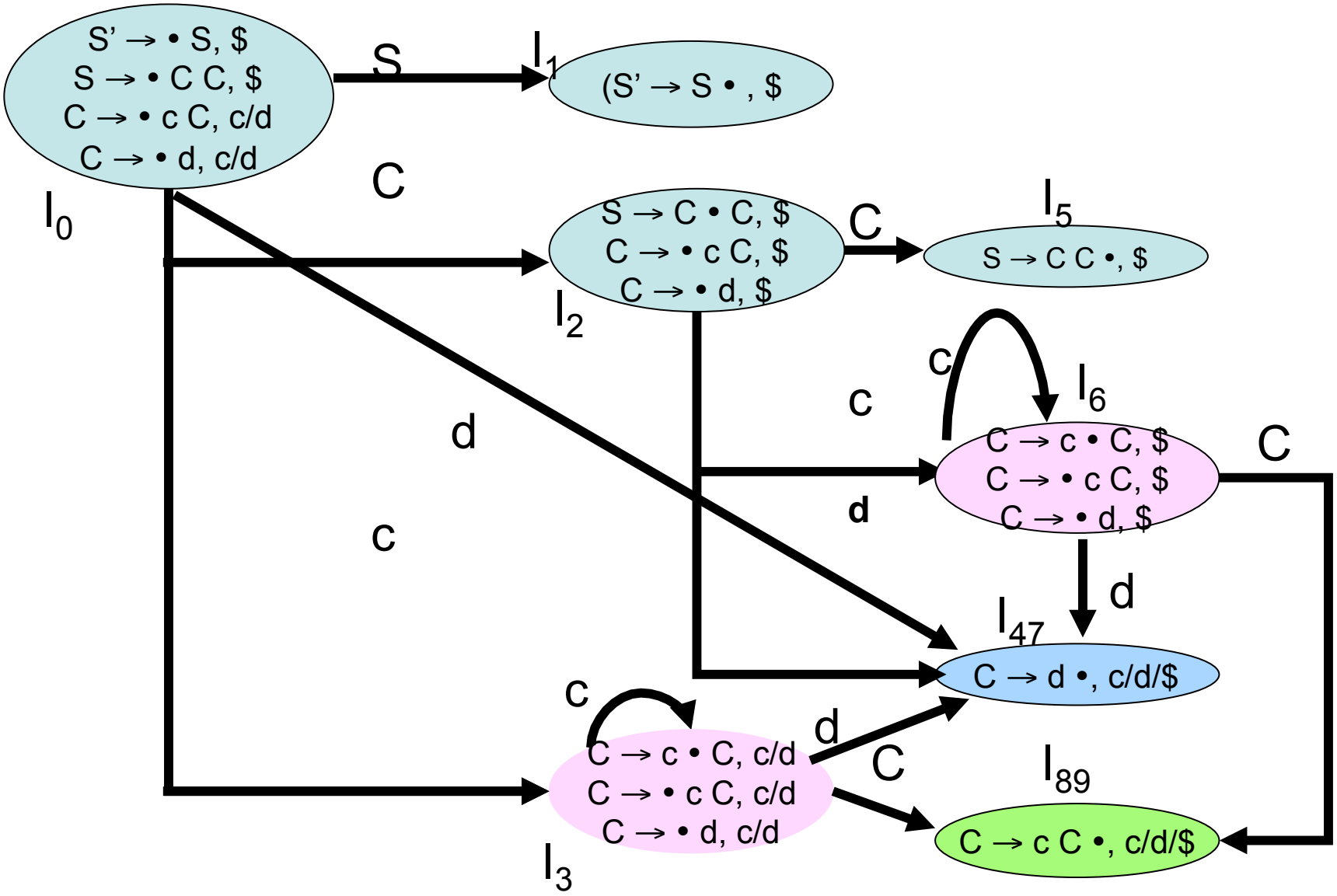
- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

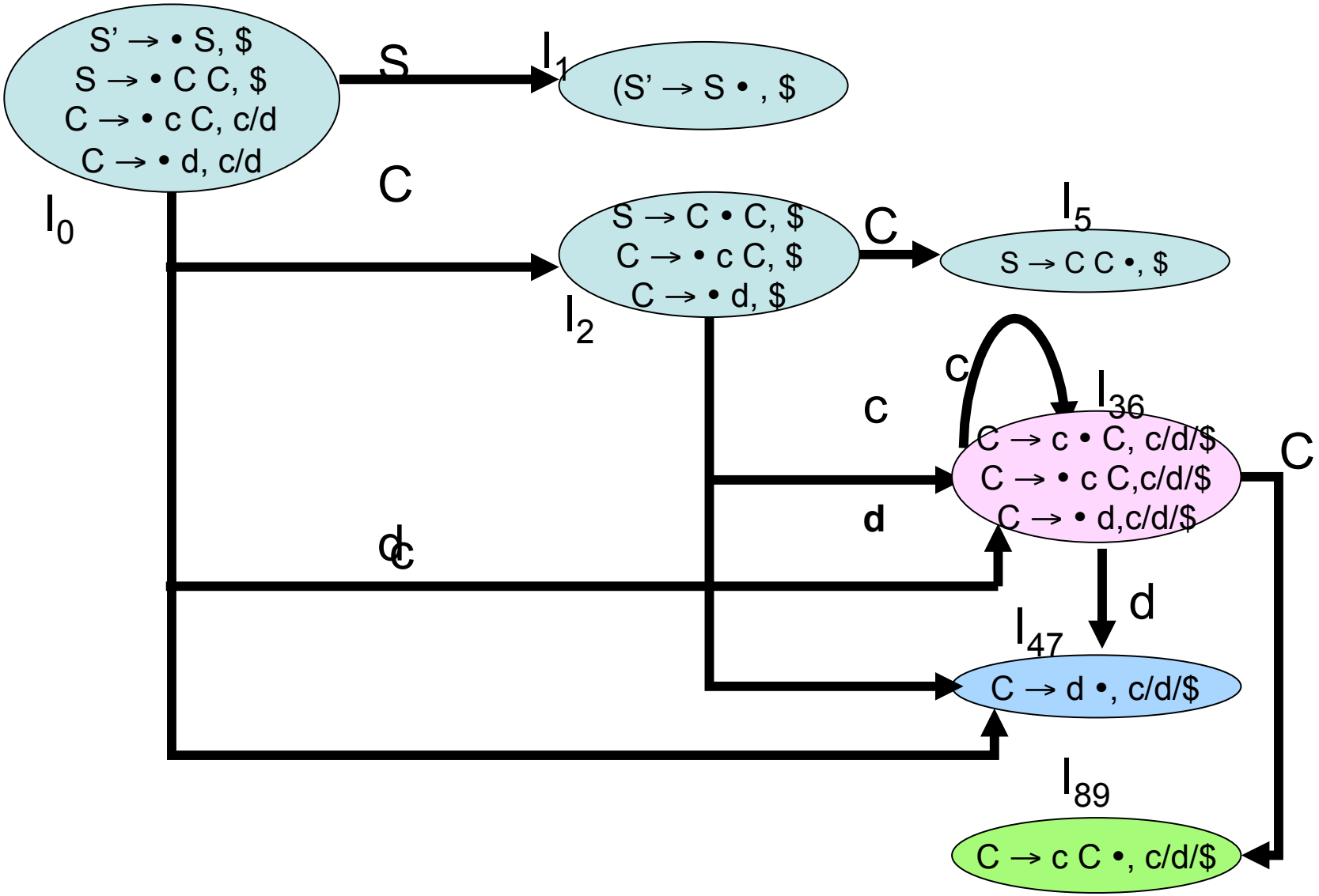
Creation of LALR Parsing Tables

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union.
   $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
   1. Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores
      ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
   1. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
4. If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
d
c
I6
C → c • C, $
C → • c C, $
C → • d, $
C
I9
C → cC •, $
d
I7
C → d •, $
c
d
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → c C •, c/d
d
I4
C → d •, c/d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
d
c
I6
C → c • C, $
C → • c C, $
C → • d, $
C
d
I7
C → d •, $
c
d
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I89
C → c C •, c/d/$
d
I4
C → d •, c/d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
d
c
c
I6
C → c • C, $
C → • c C, $
C → • d, $
C
d
I47
C → d •, c/d/$
c
d
C
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
I89
C → c C •, c/d/$


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
d
c
I36
C → c • C, c/d/$
C → • c C,c/d/$
C → • d,c/d/$
C
d
I47
C → d •, c/d/$
I89
C → c C •, c/d/$

LALR Parse Table

	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

  $A \rightarrow \alpha\,.\,,a$ and $B \rightarrow \beta\,.\,a\gamma,b$

- This means that a state of the canonical LR(1) parser must have:

  $A \rightarrow \alpha\,.\,,a$ and $B \rightarrow \beta\,.\,a\gamma,c$

  But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

  (Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

I_1 : $A \rightarrow \alpha\,.\,,a$

$B \rightarrow \beta\,.\,,b$

I_2: $A \rightarrow \alpha\,.\,,b$

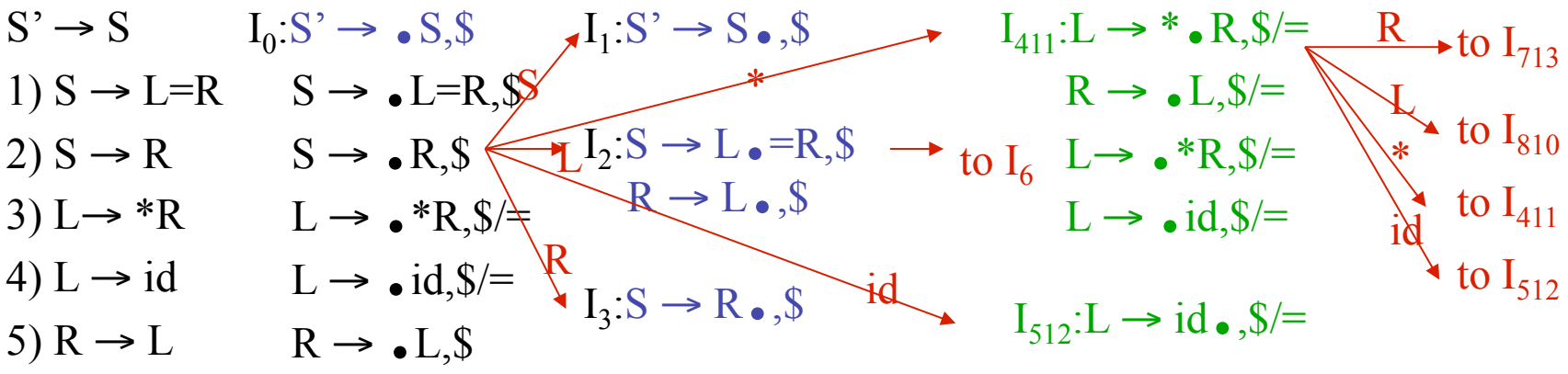
$B \rightarrow \beta\,.\,,c$

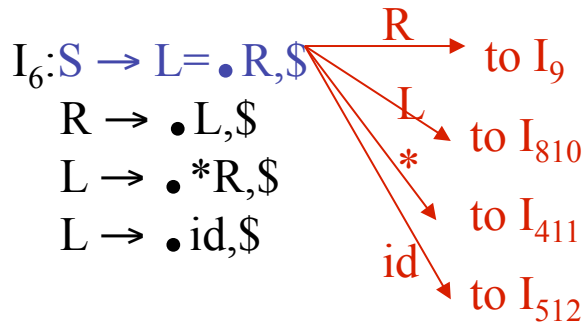
$\Downarrow$

I_{12}: $A \rightarrow \alpha\,.\,,a/b$ ➔ reduce/reduce conflict

$B \rightarrow \beta\,.\,,b/c$

Canonical LALR(1)– Ex2





I713:L → *R•,$/=

I810: R → L•,$/=

I9:S → L=R•,$

Same Cores

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing– (for Ex2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

so, it is a LALR(1) grammar

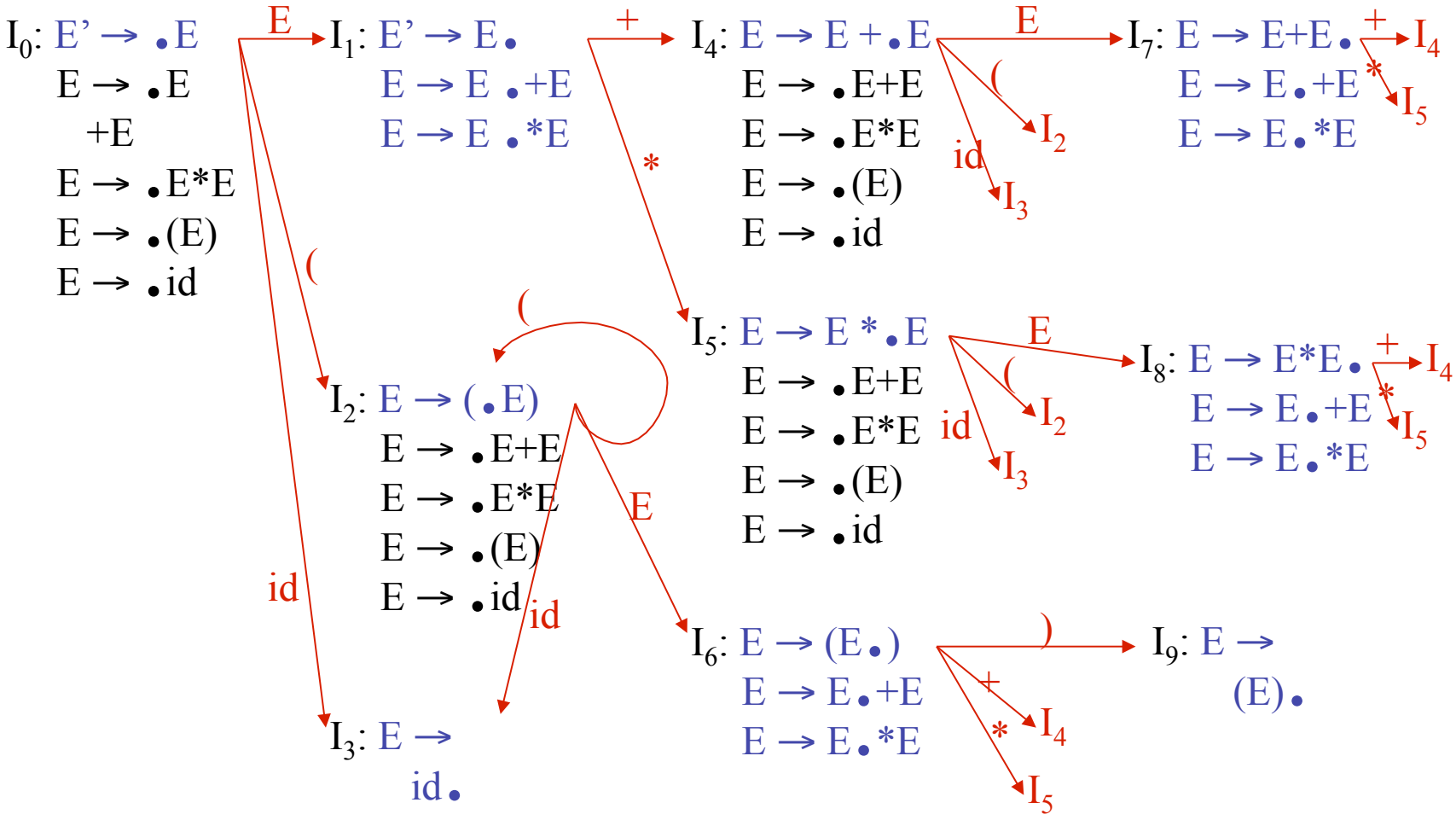
Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why use an ambiguous grammar?
  - Some of the ambiguous grammars are **more natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$E \rightarrow E+E \mid E*E \mid (E) \mid id$ ➔

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Sets for Ambiguous Grammar



SLR Tables for Amb Grammar

FOLLOW(E) = { $, +, *,) }

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +

shift ➔ + is right-associative

reduce ➔ + is left-associative

when current token is *

shift ➔ * has higher precedence than +

reduce ➔ + has higher precedence than *

SLR Tables for Amb Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_8$$

when current token is *

shift ➔ * is right-associative

reduce ➔ * is left-associative

when current token is +

shift ➔ + has higher precedence than *

reduce ➔ * has higher precedence than +

SLR Tables for Amb Grammar

	Action							Goto
	id	+	*	(	)	$		**E**
0	s3			s2				1
1		s4	s5			acc		
2	s3			s2				6
3		r4	r4		r4	r4		
4	s3			s2				7
5	s3			s2				8
6		s4	s5		s9			
7		r1	s5		r1	r1		
8		r2	r2		r2	r2		
9		r3	r3		r3	r3		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes normal parsing.
- This nonterminal A is normally a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, …

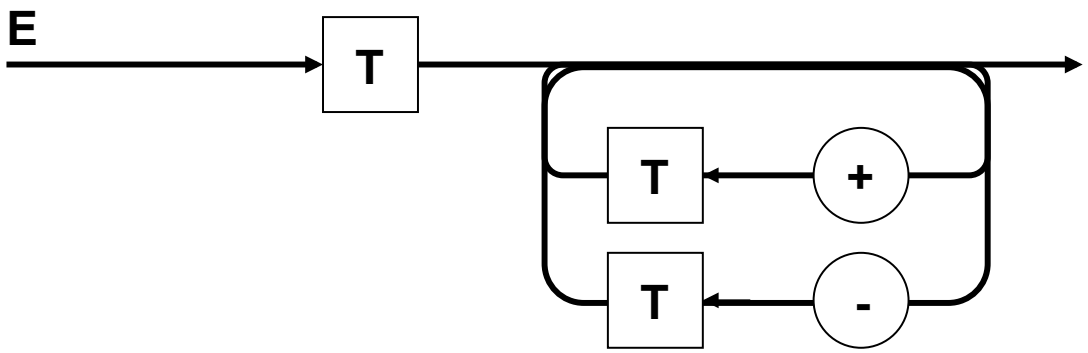
Phrase-Level Error Recovery

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

Assign#4 Help

Days#20

A Little Assign#4 Help



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

E with No Error Recovery

```
int E() {
        int op1, op2; symbol op;
        op1 := T();
        while sy in [plus, minus] do {
                op := sy;
                op2 := T();
                op1 := emit(op, op1, op2);
        }
        return op1;
}
```

Syntax Directed Translation

Days#21,22

Syntax Directed Defn. (SDD)

- A CFG with attributes and rules
- Example:
  - Production Semantic Rule
  - $E \rightarrow E_1 + T$ $E.code = E_1.code \,||\, T.code \,||\, \text{‘+’}$
- In above || is concatenation of strings
- The example shows synthesized attributes
  – ones that flow up the parse tree

SDD in Bison

- Bison/Yacc has built-in notion of attributes referred to as $$, $1, etc.
- In Bison, you can declare type YYSTYPE to override default of int used for yylval
- Often one uses a union and the type can be referenced by union type tag $<tag>$ or $<tag>1 etc.
- If you do not set $$ then default is $$ = $1

Attributes

- Attributes are synthesized when they are defined at a node labeled A using attributes of the node and its children.
- Attributes are inherited when they are defined at node labeled A using attributes of the node, its parent and its children.
- Terminals are only allowed to have synthesized attributes.

S- and L-sttributed

- An SDD with only synthesized attributes is called S-attributed. S-attributed are often used for bottom-up where they can be evaluated on the fly.
- An SDD with mixed synthesized and inherited attributes is called L-attributed if it can be evaluated left-to-right and depth first. L-attributed are often used for top-down.
- Under certain circumstances (the right kind of dependencies) both types can be evaluated in one pass.

S-attributed

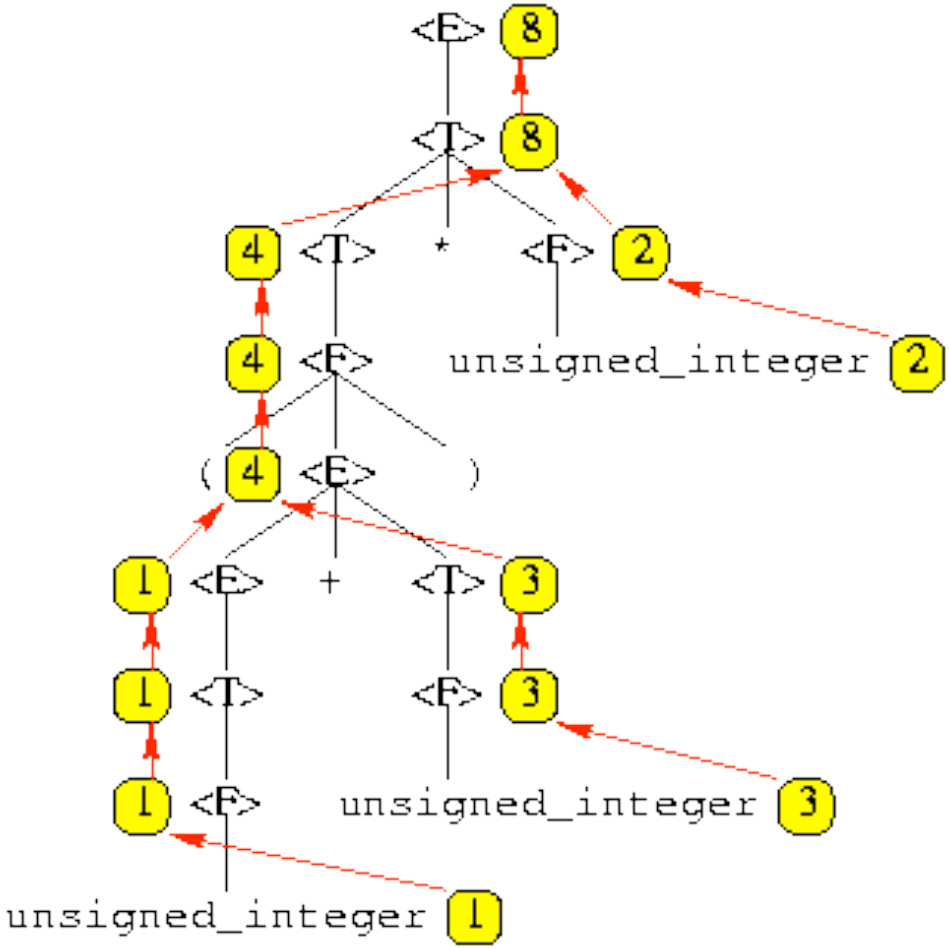
<E> → <E_1> + <T>	E.val := E_1.val + T.val
<E> → <E_1> – <T>	E.val := E_1.val – T.val
<E> → <T>	E.val := T.val
<T> → <T_1> * <F>	T.val := T_1.val * F.val
<T> → <T_1> / <F>	T.val := T_1.val / F.val
<T> → <F>	T.val := F.val
<F> → – <F_1>	F.val := – F_1.val
<F> → (<E>)	F.val := E.val
<F> → id	F.val := id.entry
<F> → unsigned_integer	F.val := unsigned_integer.val

Annotated Parse Tree

- Adds attributes to nodes
- Also called attributed or decorated
- S-attributed evaluates up the tree (typically post-order traversal but any bottom-up works)
- L-attributed evaluates pre-order and left to right. The pre-order allows attributes to flow from parent; left-to-right allows younger children to pass values along.

S-attributed Evaluation

- (1+3)*2



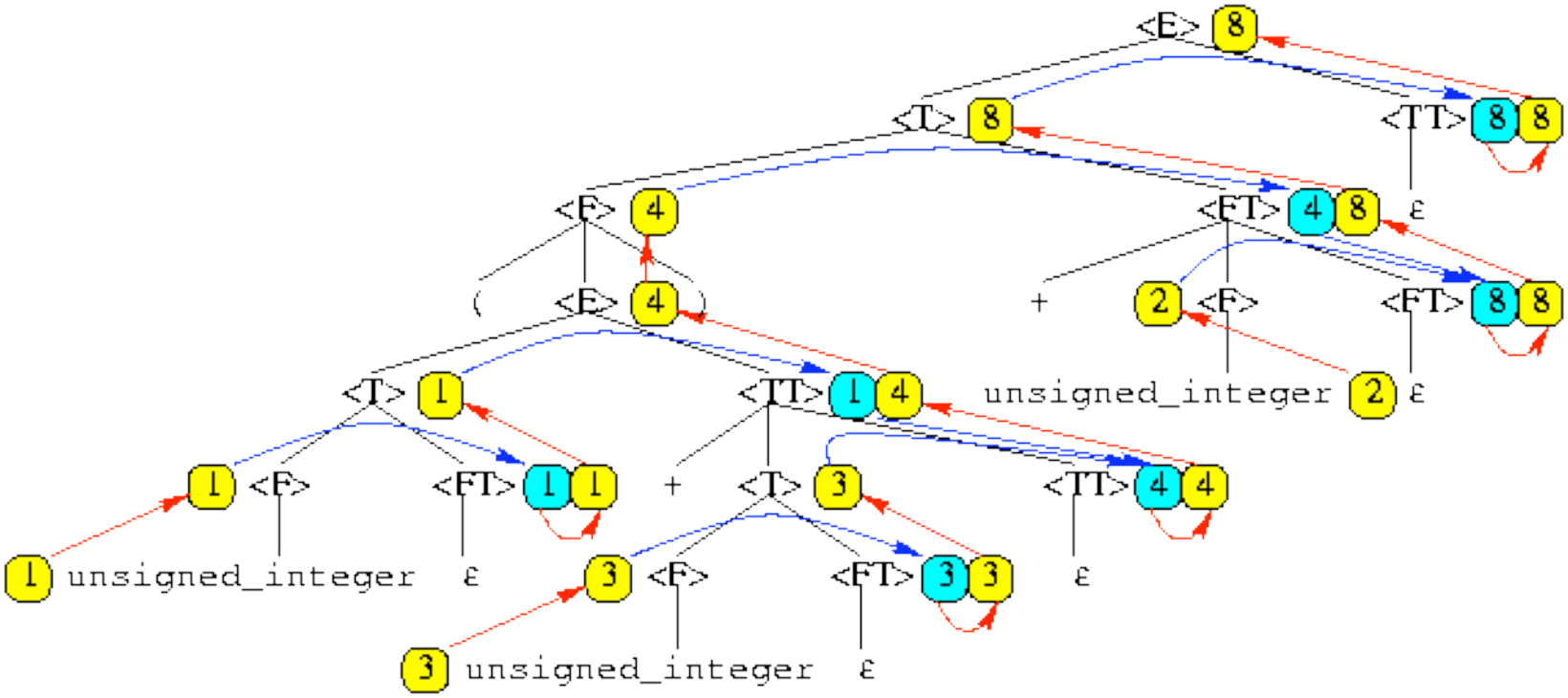
L-attributed

<E> → <T> <TT>	TT.inh := T.val; E.val := TT.syn
<TT> → + <T> <TT_1>	TT_1.inh := TT.inh + T.val; TT.syn := TT_1.syn
<TT> → – <T> <TT_1>	TT_1.inh := TT.inh – T.val; TT.syn := TT_1.syn
<TT> → ε	TT.syn := TT.inh
<T> → <F> <FT>	FT.inh := T.val; E.val := FT.syn
<FT> → * <F> <FT_1>	FT_1.inh := FT.inh × F.val; FT.syn := FT_1.syn
<FT> → / <F> <FT_1>	FT_1.inh := FT.inh / F.val; FT.syn := FT_1.syn
<FT> → ε	FT.syn := FT.inh
<F> → – <F_1>	F.val := – F_1.val
<F> → (<E>)	F.val := E.val
<F> → id	F.val := id.entry
<F> → unsigned_integer	F.val := unsigned_integer.val

Note: inherited attribute of a node can be assigned to synthesized attribute but not vice versa.

L-attributed Evaluation

- (1+3)*2 (yellow is syn or val; blue is inh)



Declaration Statements

<D> → <T> <L>	L.inh := T.type
<T> → int	T.type := integer
<T> → float	T.type := float
<L> → <L_1> ‘,’ id	L_1.inh := L.inh; addType(id.entry,L.inh)
<L> → id	addType(id.entry,L.inh)

Evaluation Order

- Any order that maintains dependencies is acceptable in attribute evaluation
- Typical approach is topological sort of dependency graph
- Problem: If actions have side effects then one must be careful to not change semantics with varying orders of evaluation

S-synthesizing a Syntax Tree

<E> → <E_1> + <T>	E.node := new Node('+', E_1.node, E.node)
<E> → <E_1> – <T>	E.node := new Node('–', E_1.node, E.node)
<E> → <T>	E.node := T.node
<T> → <T_1> * <F>	T.node := new Node('*', T_1.node, F.node)
<T> → <T_1> / <F>	T.node := new Node('/', T_1.node, F.node)
<T> → <F>	T.node := F.node
<F> → – <F_1>	F.node := new UnaryNode(minus, F_1.node)
<F> → (<E>)	F.Node := E.node
<F> → id	F.node := new LeafNode(ident, id.entry)
<F> → num	F.node := new LeafNode(number, num.val)

L-synthesizing a Syntax tree

<E> → <T> <TT>	E.node := TT.syn; TT.inh := T.node
<TT> → + <T> <TT_1>	TT_1.inh := new Node('+',TT.inh,T.node; TT.syn := TT_1.syn
<TT> → – <T> <TT_1>	TT_1.inh := new Node('–',TT.inh,T.node; TT.syn := TT_1.syn
<TT> → ε	TT.syn := TT.inh
<T> → <F> <FT>	T.node := FT.syn; FT.inh := F.node
<FT> → * <F> <FT_1>	FT_1.inh := new Node('×',FT.inh,F.node; FT.syn := FT_1.syn
<FT> → / <F> <FT_1>	TT_1.inh := new Node('/',TT.inh,F.node; FT.syn := FT_1.syn
<FT> → ε	FT.syn := FT.inh
<F> → – <F_1>	F.val := – F_1.val
<F> → (<E>)	F.node := E.node
<F> → id	F.node := new LeafNode(ident, id.entry)
<F> → num	F.node := new LeafNode(number, num.val)

Flattening of the Syntax Tree

- Triples and quads as we defined them area form of flattening
- Triples are compact but hard to move
- Quads are wasteful in many cases, but easy to move, e.g., from inside a loop to precede it when semantics are still correct

Indirect Triples

- Compromise between triples and quads
- Generate triples but have a separate list that specifies which triples actually are at a particular node position

Dataflow Analysis

Days#25,26,27

Dataflow Analysis

- Use of data flow within program to determine producer/consumer relationship between points in program.

- Information sought at each point in program
  - Where were values produced that might be consumed here?
  - Where are values consumed that are produced here?
  - What are constraints on values available here?

Scalar Analysis

- Basic type is Scalar Analysis
  - Concentrates on simple variable names
  - Indexed array ref. A[I] is treated as a reference to all of object A
  - This basic coverage ignores aliasing (multiple names for same object)

Focus of Analysis

- Basic Block
  - One in, one out sequence of code
- Local Analysis – done on single basic blocks
- Intraprocedural Analysis – done within procedures
- Interprocedural Analysis – done across procedures

Control vs Data Flow

- Control Flow
  - intra creates flow graph with procedure entry as initial node
  - inter creates a call graph with main body as initial node
- Data Flow
  - determines accessibility of definitions and uses to each other
  - UD chaining
    - given a variable use, what definitions reach this use
  - DU chaining
    - given a variable definition, what uses are made of it

Program representation

- Control Flow Graph
  - Nodes N – statements of program (maybe of intermediate code; maybe of source code)
  - Edges E – flow of control
    - pred(n) = set of all predecessors of n
    - succ(n) = set of all successors of n
  - Start node n_0
  - Set of final nodes N_{final}

Control Flow Graph

- Program P consists of procedures, one of which is denoted p.
- We assume one entry / one exit procedures.
- A control flow graph G = (N, E, s) refers to a directed graph (N, E) and an initial node s in N, where there is a path from s to every node of G.
- Nodes can be statements or basic blocks. Commonly, they are the latter.

Basic Blocks Example

```
Program SquareRoot;
var   L, N, K, M : integer; C : boolean;
begin
      (* start of block B1 *)
      read(L);
      N ← 0;
      K ← 0;
      M ← 1;
      (* end of block B1 *)
      loop
            (* start of block B2 *)
            K ← K + M;
            C ← K > L;
            if C then break;
            (* end of block B2 *)
            (* start of block B3 *)
            N ← N + 1;
            M ← M + 2
            (* end of block B3 *)
      end loop;
      (* start of block B4 *)
      write(N)
      (* end of block B4 *)
end.  (* SquareRoot *)
```

Program Points

- One program point before each node
- One program point after each node
- Join point – point with multiple predecessors
- Split point – point with multiple successors

Basic Idea

- Information about program represented using values from algebraic structure called lattice
- Analysis produces lattice value for each program point
- Two flavors of analysis
  - Forward dataflow analysis
  - Backward dataflow analysis

Forward Dataflow Analysis

- Analysis propagates values forward through control flow graph with flow of control
  - Each node has a transfer function f
    - Input – value at program point before node
    - Output – new value at program point after node
  - Values flow from program points after predecessor nodes to program points before successor nodes
  - At join points, values are combined using a merge function
- Canonical Example: Reaching Definitions

Backward Dataflow Analysis

- Analysis propagates values backward through control flow graph against flow of control
  - Each node has a transfer function f
    - Input – value at program point after node
    - Output – new value at program point before node
  - Values flow from program points before successor nodes to program points after predecessor nodes
  - At split points, values are combined using a merge function
- Canonical Example: Live Variables

Partial Orders

- Set P
- Partial order $\leq$ such that $\forall x,y,z \in P$
  - $x \leq x$ (reflexive)
  - $x \leq y$ and $y \leq x$ implies $x = y$ (asymmetric)
  - $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)

Upper Bounds

- If $S \subseteq P$ then
  - $x \in P$ is an upper bound of S if $\forall y \in S.\ y \leq x$
  - $x \in P$ is the least upper bound of S if
    - x is an upper bound of S, and
    - $x \leq y$ for all upper bounds y of S
  - $\vee$ - join, least upper bound
    - $\vee$ S is the least upper bound of S
    - $x \vee y$ is the least upper bound of {x,y}

Lower Bounds

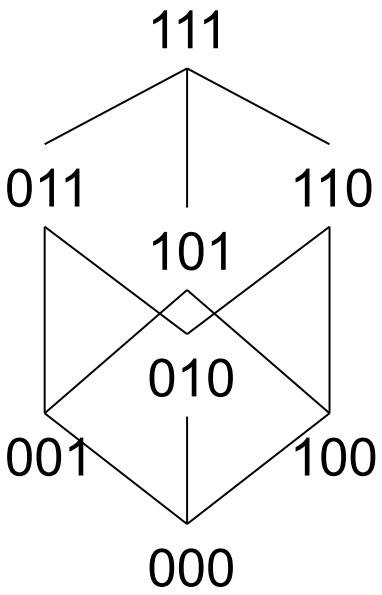
- If $S \subseteq P$ then
  - $x \in P$ is a lower bound of S if $\forall y \in S.\ x \leq y$
  - $x \in P$ is the greatest lower bound of S if
    - x is a lower bound of S, and
    - $y \leq x$ for all lower bounds y of S
  - $\wedge$ - meet, greatest lower bound
    - $\wedge$ S is the greatest lower bound of S
    - $x \wedge y$ is the greatest lower bound of {x,y}

Covering

- $x < y$ if $x \leq y$ and $x \neq y$
- x is covered by y (y covers x) if
  - $x < y$, and
  - $x \leq z < y$ implies $x = z$
- Conceptually, y covers x if $x < y$ and there are no elements between x and y

Example

- P = { 000, 001, 010, 011, 100, 101, 110, 111}
  (standard boolean lattice, also called hypercube)
- x ≤ y if (x bitwise and y) = x



Hasse Diagram

- If y covers x
  - Line from y to x
  - y above x in diagram

Lattices

- If $x \wedge y$ and $x \vee y$ exist for all $x,y \in P$,
  then P is a lattice.
- If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$,
  then P is a complete lattice.
- All finite lattices are complete
- Example of a lattice that is not complete
  - Integers I
  - For any $x, y \in I$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
  - But $\vee$ I and $\wedge$ I do not exist
  - $I \cup \{+\infty,-\infty\}$ is a complete lattice

Top and Bottom

- Greatest element of P (if it exists) is top
- Least element of P (if it exists) is bottom ($\bot$)

Extracting Loops

- Let G = (N,E,s)
  - A node s' ∈ N is the entry point for loop in G iff there is an s" ∈ N such that (s", s') ∈ E and s' dominates s". (s' dominates s" if s' is on every path from s (start node) to s")
  - Let s' be an entry point of a loop. The max loop with entry s' is G' = (N',E',s'), where
    N' = {s" | ∃ a path from s" to s' containing only nodes "dominated" by s'}.
    E'=E ∩ (N'×N')
- To do data flow analysis we often wish to obey dominances, doing loop entries before their bodies, if conditions before their choices, etc.

Numbering Nodes

- A depth first traversal can be used to number nodes so that
- s' < s" (s' dominates s") implies #(s') < #(s").
- Note that it is not true that #(s') < #(s") implies s' < s".

DFT Algorithm

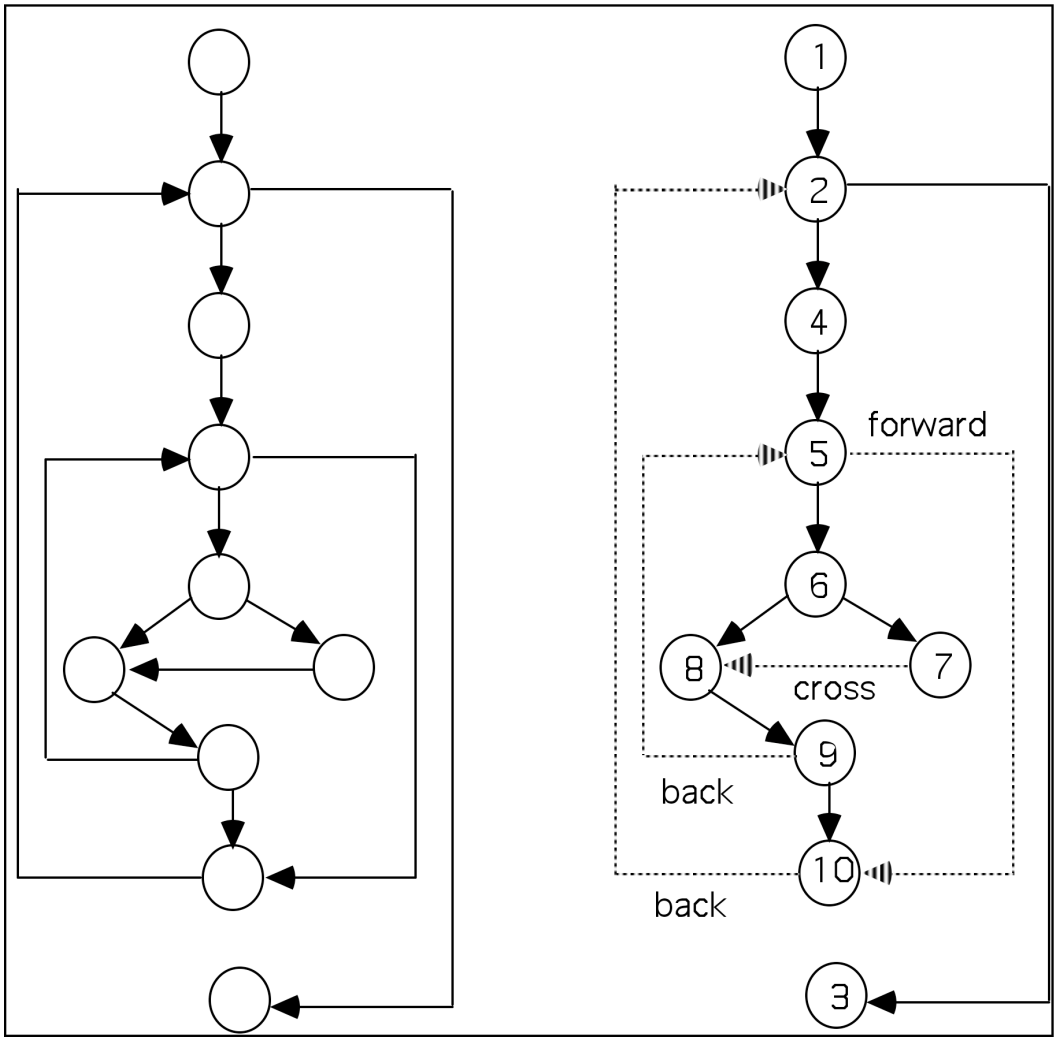
```
DFT( G : flowgraph )                    (* G = (N,E,s) *)
    E' ← { };
    i ← | N |;
    for every t in N do t.mark ← false;
    search( s )

Search( t : node )
    t.mark ← true;
    while t.unmarked_successors ≠ { } do begin
            t' ← select( t.unmarked_successor );
            E' ← E' + { (t,t') };
            Search( t' )
    end; (* while *)
    rPostOrder[t] ← i;
    i ← i – 1
```

Properties of DFT

- This produces one of the natural orders. Visiting nodes based on these numbers speeds up data flow analysis.
- Arcs are forward (unvisited node); back (visited but not numbered); cross (numbered).
- Back arcs denote loops.

Categorizing Arcs in DFT



An Aside: Managing Heap

- DFT can be used in managed language
- Start with handles known to be accessible (use some root that gets all of these)
- Follow handles (pointers) to see what is accessible

Putting Pieces Together

- Forward Dataflow Analysis Framework
- Simulates execution of program forward with flow of control

Forward Dataflow Analysis

- Simulates execution of program forward with flow of control
- For each node n, have
  - in_n – value at program point before n
  - out_n – value at program point after n
  - f_n – transfer function for n (given in_n, computes out_n)
- Require that solution satisfy
  - $\forall n.\ out_n = f_n(in_n)$
  - $\forall n \neq n_0.\ in_n = \vee \{ out_m \mid m \text{ in } pred(n) \}$
  - $in_{n0} = I$
  - Where I summarizes information at start of program

Dataflow Equations

- Compiler processes program to obtain a set of dataflow equations

  $out_n := f_n(in_n)$

  $in_n := \vee \{ out_m \mid m \text{ in } pred(n) \}$

- Conceptually separates analysis problem from program

May versus Must Analysis

- We now have a recurrence relation and hence seek a fixed point.
- We want the best but correct fixed point.

- MAY – determine if a property may be possible. This is attacked by assuming no elements satisfy, then union in all those that might have the property. By starting with the empty set, we get the Least Upper Bound (LUB). This is conservative.
- MUST – determine if a property must be true. This is attacked by assuming all elements satisfy, then intersecting all those that must have the property. By starting with everything, we get the Greatest Lower Bound (GLB). This is conservative.

Direction of Flow

- FORWARD FLOW – information flows from the root towards leaves of the control flow graph.
- BACKWARD FLOW – information goes from the leaves towards the root of the control flow graph.

Alg for Forward Dataflow

for each n do $out_n := f_n(\{\}) = f_n(\bot)$
$in_{n0} := \{\}$; // or available at start
$out_{n0} := f_{n0}(in_{n0})$
worklist := N - { n_0 }
while worklist $\neq \emptyset$ do
    remove a node n from worklist
    $in_n := \vee \{ out_m \mid m \text{ in pred(n)} \}$
    $out_n := f_n(in_n)$
    if out_n changed then
        worklist := worklist $\cup$ succ(n)

Correctness Argument

- Why result satisfies dataflow equations
- Whenever process a node n, set $out_n := f_n(in_n)$
  Algorithm ensures that $out_n = f_n(in_n)$
- Whenever out_m changes, put succ(m) on worklist.
  Consider any node $n \in succ(m)$. It will eventually come off worklist and algorithm will set

  $$in_n := \vee \{ out_m \,.\, m \text{ in } pred(n) \}$$

  to ensure that $in_n = \vee \{ out_m \,.\, m \text{ in } pred(n) \}$
- So final solution will satisfy dataflow equations

Reaching Definitions

- Useful in optimizations such as constant propagation and copy propagation
- For each basic block we determine the set of definitions that reach the beginning of that basic block called *in*[] and the set of definitions that reach the end of that block called *out*[]
- A definition *d* reaches a point *p* if there is a path from *d* to *p* such that *d* is not “killed” along that path

RD Abstraction

Form of data flow equations for reaching definitions is

out[S] = *gen*[S] ∪ (*in*[S] - *kill*[S])

Alternately can intersect in with preserve[S]

gen[] -- the set of definitions that reach the end of S independent of whether they reach the beginning of S

kill[] -- the set of definitions that never reach the end of S even if they reach the beginning

Reaching Definitions

- P = powerset of set of all definitions in program (all subsets of set of definitions in program)
- $\vee = \cup$ (order is $\subseteq$)
- $\perp = \varnothing$
- $I = in_{n0} = \perp$
- F = all functions f of the form $f(x) = a \cup (x-b)$
  - b is set of definitions that node kills
  - a is set of definitions that node generates
- General pattern for many transfer functions
  - $f(x) = GEN \cup (x-KILL)$

Reaching Definitions

- Notation: For any node n, pred[n] is the set of all immediate predecessors of n and succ[n] is the set of all immediate successors.
- RD[n] = ReachIn[n] = { s | p ∈ pred[n] and s ∈ ReachOut[p] }
- ReachOut[n] = (ReachIn[n] ∩ S_PRE[n]) ∪ S_DEF[n]

Implementing RD

```
for i = 1 to NBlocks do begin
    ReachOut[i] ← S_DEF[i];
    ReachIn[i] ← { }
end;
change ← true;
while change do begin
    change ← false;
    for i = 1 to NBlocks do begin
        newIn ← { s | p ∈ pred[n] & s ∈ ReachOut[p] };
        if ReachIn[i] ≠ newIn then begin
            ReachIn[i] ← newIn;
            oldOut ← ReachOut[i];
            ReachOut[i] ← (ReachIn[i] ∩ PRE[i]) ∪ GEN[i];
            /* or (ReachIn[i] - KILL[i]) ∪ GEN[i];
            if oldOut ≠ ReachOut[i] then change := true
        end
    end
end
```

Backwards dataflow

- Simulates execution of program backward against the flow of control
- For each node n, have
  - in_n – value at program point before n
  - out_n – value at program point after n
  - f_n – transfer function for n (given out_n, computes in_n)
- Require that solution satisfies
  - $\forall n.\ in_n = f_n(out_n)$
  - $\forall n \notin N_{final}.\ out_n = \vee \{ in_m \ .\ m \text{ in succ}(n) \}$
  - $\forall n \in N_{final} = out_n = O$
  - Where O summarizes information at end of program

Alg for Backward Dataflow

for each n do $in_n := f_n(\perp)$

for each $n \in N_{final}$ do $out_n := O$; $in_n := f_n(O)$

worklist := $N - N_{final}$

while worklist $\neq \emptyset$ do

    remove a node n from worklist

    $out_n := \vee \{ in_m \,.\, m \text{ in succ}(n) \}$

    $in_n := f_n(out_n)$

    if in_n changed then

        worklist := worklist $\cup$ pred(n)

Liveness

- Calculates liveness information
- A variable is said to be *live* at a point if there are further uses of that variable. Otherwise it is said to be *dead*
- Backward analysis – *ie* we move in the *backward* direction
- Still a *may* problem

Liveness Abstraction

Data Flow equations for live variable analysis

$$out[B] = \bigcup_{Succ[B]} in[S]$$

$$in[B] = use[B] \cup (out[B] - def[B])$$

use[B] = the set of variables that are used prior to any definition

def[B] = the set of variables that are definitely assigned prior to any use

Scalar dependence

S1: A ← 1.0;

S2: B ← A + 3.1415;

S3: A ← .333 * (C – D);

… …

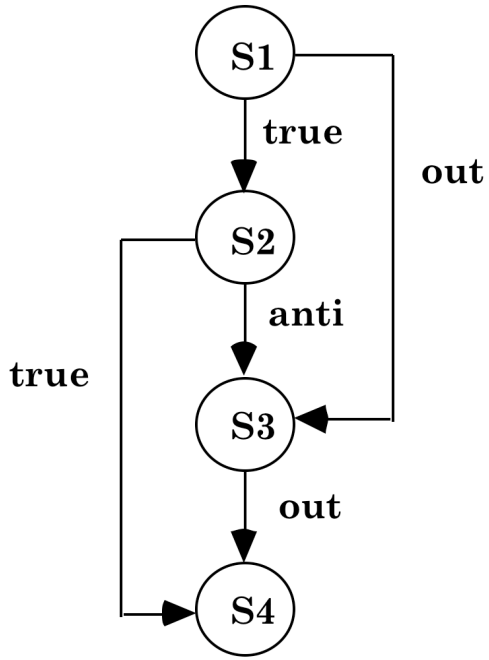
S4: A ← (B * 3.8) / 2.718;

S2 is true dependent on S1

S3 is anti-dependent on S2

S4 is output dependent on S3

Example of dependence



Final Exam Promises

- An expression grammar that incorporates precedence and associativity.
- Distinction between languages and grammars in a particular class.
- Ambiguity
- FLEX type answer to a regular expression problem.
- EBNF / Railroad chart question
- Creation of a recursive descent parser for some simple construct.
- Creation of FIRST, FOLLOW and an LL(1) parse table.
- Removal of left recursion and common prefixes.
- CKY
- Bottom-Up and Top-Down stack manipulation
- Adding actions to Bison grammar, e.g., code generation, semantic error checks
- Completion of the states, actions and gotos for an SLR(1) parser.
- Completion of canonical LR(1) parser.
- LALR(1) parser by doing merges on a canonical LR(1) parser's states.
- Evaluation of attributes (inherited and synthesized) for some attributed translation grammar.
- Data flow algorithm based on one of the four discussed in class.