Automaton versus Scanner

• Automaton accepts or rejects one word

♦ Runs until it exhausts the input and accepts or rejects the stream

• Scanner looks at the whole program and returns all of the tokens

♦ Must break the input stream into separate words

♦ Must capture and classify each lexeme

♦ Must decide when it has looked beyond the end of a word

Recognizing Word Boundaries

Two obvious solutions

• Require delimiters between every token, which is ugly and painful

• Run the automaton to an error or EOF, and then back up to a final state

Scanner

• Specify syntax with regular expressions (REs)

• Construct finite-automaton & scanner from the RE Parser

• Specify syntax with context-free grammars (CFGs)

• Construct push-down automaton & parser from the CFG

A CFG is a four tuple, **G = (S,N,T,P)**

• S is the start symbol of the grammar

L(G) is the set of sentences that can be

derived from S

• N is a set of nonterminal symbols or

syntactic variables { Goal, List, Pair }

• T is a set of terminal symbols or words

{ (, ) }

• P is a set of productions or rewrite rules,

as shown in the table to the left

A derivation consists of a series of rewrite steps : Symbol to Sentence

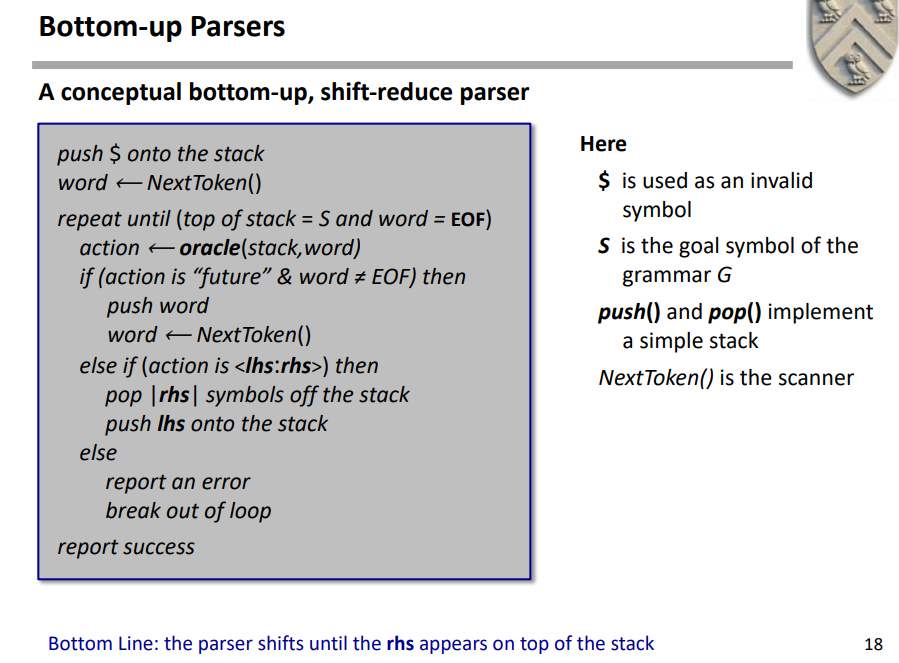
♦ Replacing the leftmost NT at each step, creates a leftmost derivation

♦ Replacing the rightmost NT at each step, creates a rightmost derivation

The point of parsing is to discover a grammatical derivation for a sentence

Bottom-up parsers can recognize a larger class of grammars than can top-down parsers

LR1 : 1 word of lookahead



A shift-reduce parser has four kinds of actions:

Shift: next word is moved from input to the stack

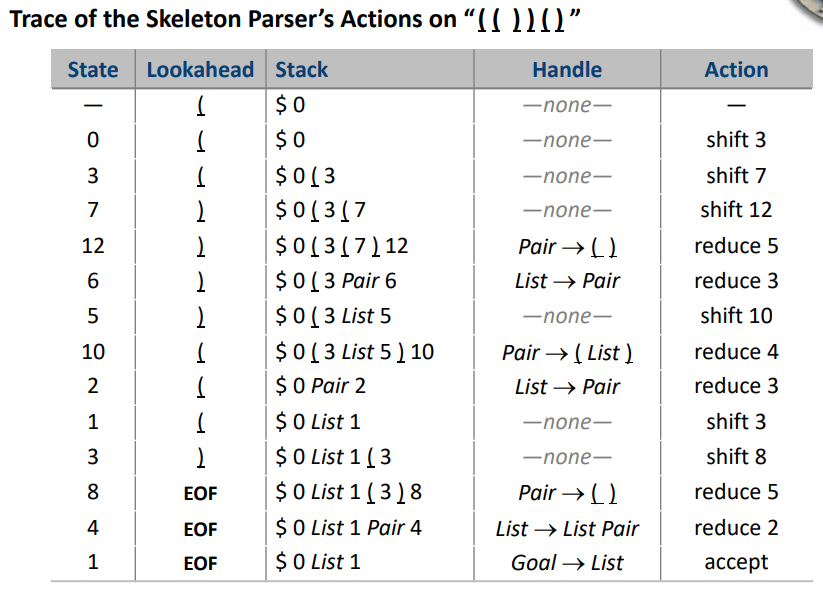
Reduce: TOS is rhs of a reduction pop rhs off the stack, push lhs onto the stack

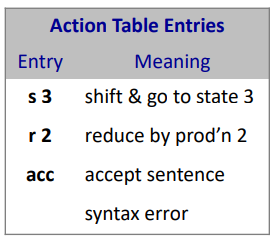
Error: report the problem to user

Accept: (normal exit from loop) report success and stop parsing

Shift, Accept, & Error are O(1) Reduce is O(|rhs|) ∑rhs |rhs| = |nodes in parse tree|

the oracle is encoded into two parse tables: ACTION and GOTO





Shift查action表 转state， reduce 查 goto（根据前一个state行，rule的NT列）和rule表 转state

How do I get my CFG into a form where it is parsable with LR(1) techniques?

• The grammar must be unambiguous (deterministic)

• The grammar must be parsable with single word lookahead

Ambiguous: 多个最左推导/多个最右推导/最左最右结果不同（parse tree不同）

Rewrite: Expr -> Expr Op Expr | int | float;

To: Expr -> Value Op Expr | Value

Some important IR properties

• Ease of generation

• Ease of manipulation

• Cost of manipulation

• Procedure size

• Expressiveness

• Level of abstraction

Is the IR closer in abstraction to the source or the machine?

♦ “High level” ⇒ closer to source

♦ “Low level” ⇒ closer to machine

Three major categories of IR

• Structural IRs: AST, DAG

♦ Graphs and trees

♦ Widely used in source-to-source translators

♦ Tend to use large amounts of memory

• Linear IRs: ILOC, RTL, Stack Machine Code (One-address code)

♦ Pseudo-code for an abstract machine

♦ Simple compact data structures

♦ Can be easy to reorder and rearrange

SMC:

x - 2 \* y becomes

push 2

push y

Multiply

push x

subtract

In general, three-address code has the form

x <- y op z

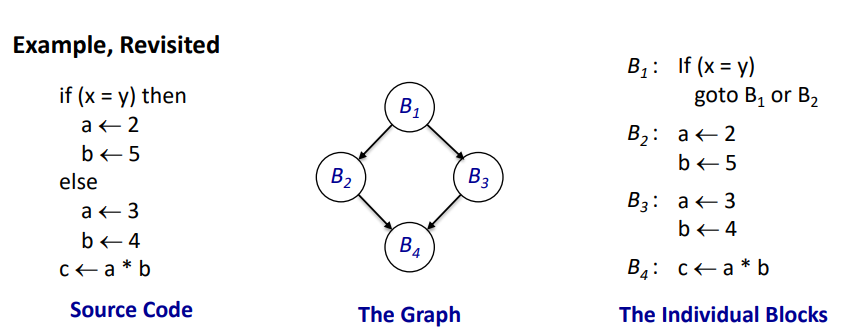
with at most one operator (op) and three names (x, y, & z)

• Hybrid IRs: CFG, SSA form

♦ Combinations of graphs and linear code

♦ Provide some of the advantages of both structural & linear IRs

CFG:



SSA:

each name is defined by exactly one operation

A compiler’s IR also includes several tables and maps:

Symbol table, constant table, storage table.

• Ambiguous values must live in memory

• Any unambiguous value is a candidate for a register

Array Layout

Row-Major Order

• Lay out as a sequence of consecutive rows

• Rightmost subscript varies fastest

• A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-Major Order

• Lay out as a sequence of columns

• Leftmost subscript varies fastest

• A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

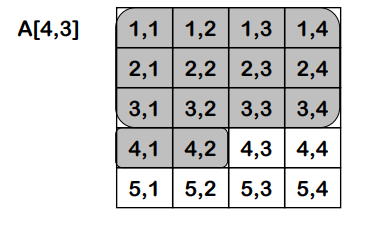
Indirection Vectors

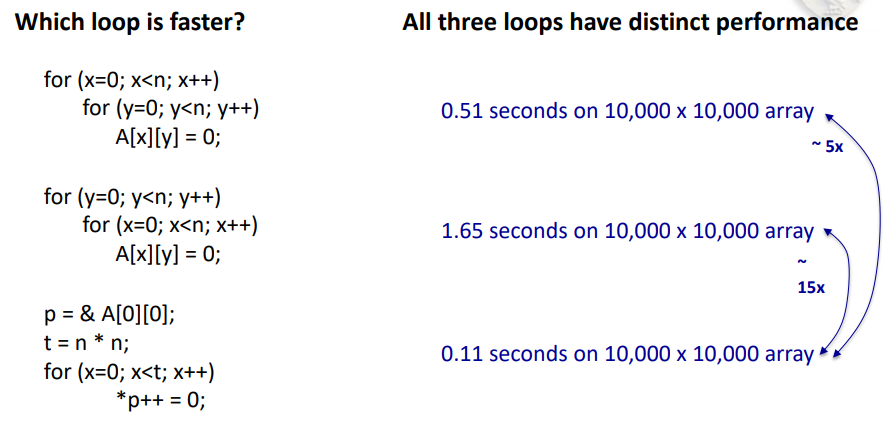
• Vector of pointers to pointers to … to values

• Takes much more space, trades indirection for arithmetic

• Not amenable to analysis

Address of A[i]: base(A) + ( i – low ) x sizeof(A[1])





Data that represents an invocation

• Activation record (AR)

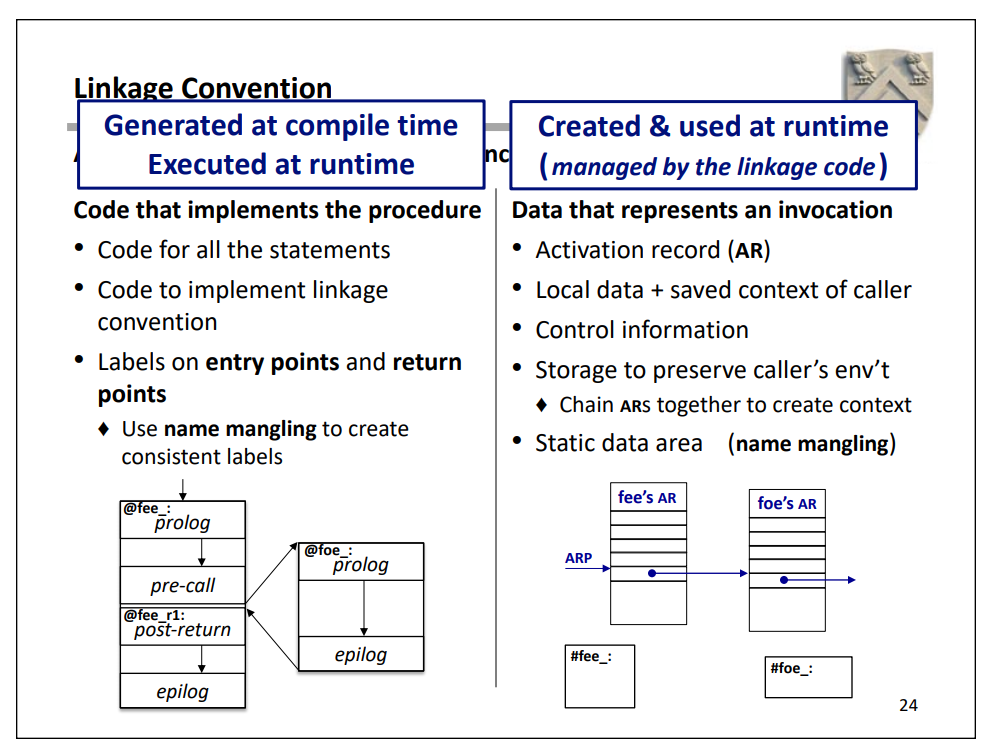
• Local data + saved context of caller

• Control information

• Storage to preserve caller’s env’t

♦ Chain ARs together to create context

• Static data area (name mangling)



Meta-Issue: Activation records are created, used and destroyed at runtime. The code to maintain them is emitted at compile time.

These four code sequences:

1. Allocate and initialize a new AR for the callee

2. Populate that AR with needed values & context

3. Preserve parts of the caller’s environment that might be modified by callee

– Contents of registers, pointer to its activation record, …

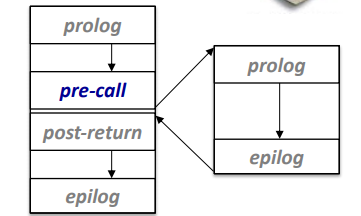
4. Handle parameter passing

– Evaluate parameters in caller

– Make them available in callee

5. Transfer control from caller to callee & back (LIFO behavior)

ARs are runtime structures, laid out at compile time



Pre-call Sequence

• Sets up callee’s basic AR

• Helps preserve its own environment

Post-return Sequence

• Finish restoring caller’s environment

• Place any value back where it belongs

♦ e.g., reference parameter or global that kept in a register in the callee

Epilog Code

• Wind up (Finish) the business of the callee

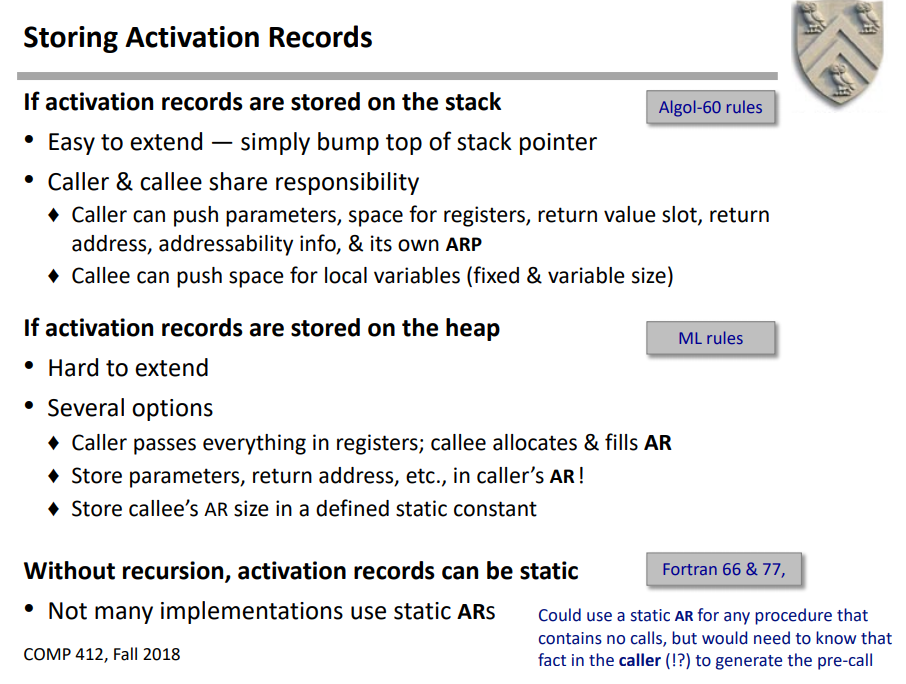
• Start restoring the caller’s environment

Caller & Callee’s collaboration on AR:

1. Linkage convention is defined at compiler design time

2. Linkage code is emitted at compile time

3. Linkage code executes at runtime, when a procedure is called



Automatic: Lifetime matches procedure activation

Static: Lifetime may be as long as entire execution

Dynamic: Lifetime is under program control & notknown at compile time

Lexically-Scoped Symbol Tables

High-level idea

• Create a new table for each scope

• Chain them together for lookup

Typically, the programming language maps declarations to lifetimes

• Bad layout can waste space (padding)

Fortunately, it is easy to minimize padding

• Order the variables into groups, from most restrictive alignment to least

• Assign to all items in a group, then move to the next group

• Typically, this approach limits padding to group boundaries

