CS 132 Compiler Construction

Chapter 1: Introduction

Things to do

- make sure you have a working SEAS account
- start brushing up on Java
- review Java development tools
- find http://www.cs.ucla.edu/ palsberg/courses/cs132/F03/index.html
- check out the discussion forum on the course webpage

Compilers

What is a compiler?

- a program that translates an *executable* program in one language into an *executable* program in another language
- we expect the program produced by the compiler to be better, in some way, than the original

What is an interpreter?

- a program that reads an *executable* program and produces the results of running that program
- usually, this involves executing the source program in some fashion

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters*

Motivation

Why study compiler construction?

Why build compilers?

Why attend class?

Interest

Compiler construction is a microcosm of computer science

artificial intelligence	greedy algorithms learning algorithms
algorithms	graph algorithms union-find dynamic programming
theory	DFAs for scanning parser generators lattice theory for analysis
systems	allocation and naming locality synchronization
architecture	pipeline management hierarchy management instruction set use

Inside a compiler, all these things come together

Isn’t it a solved problem?

Machines are constantly changing

Changes in architecture $\Rightarrow$ changes in compilers

- new features pose new problems
- changing costs lead to different concerns
- old solutions need re-engineering

Changes in compilers should prompt changes in architecture

- New languages and features

Intrinsic Merit

Compiler construction is challenging and fun

- interesting problems
- primary responsibility for performance (*blame*)
- new architectures $\Rightarrow$ new challenges
- *real* results
- extremely complex interactions

Compilers have an impact on how computers are used

Compiler construction poses some of the most interesting problems in computing

Experience

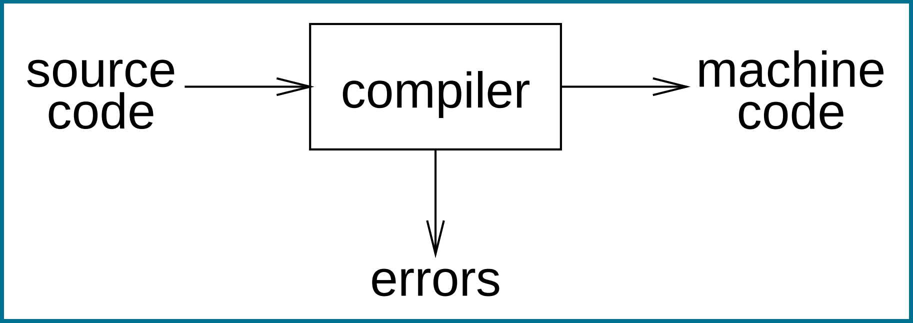
You have used several compilers

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your feelings about the correct contents of this course

Abstract view

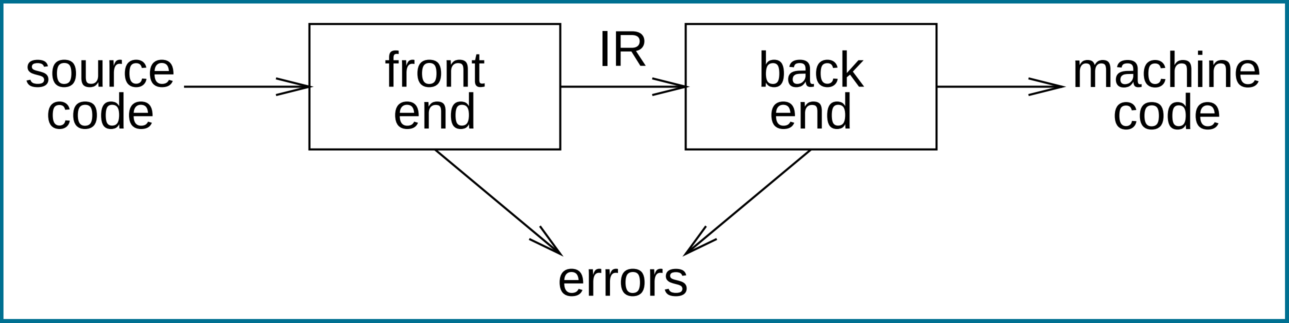


Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

Big step up from assembler — higher level notations

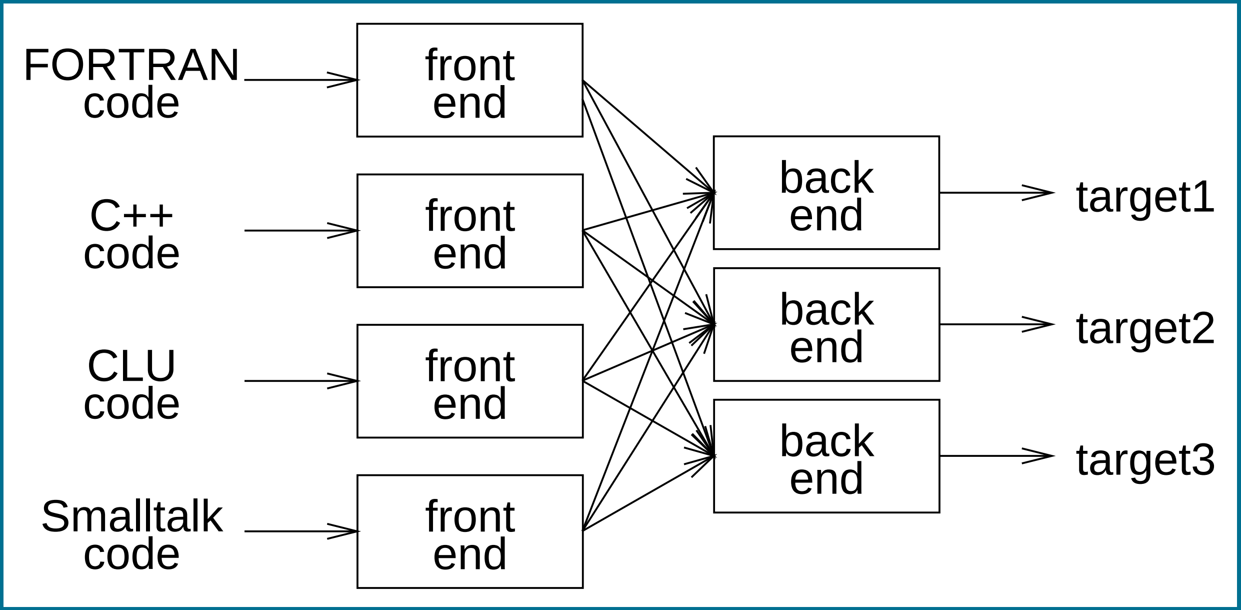
Traditional two pass compiler



Implications:

- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

A fallacy

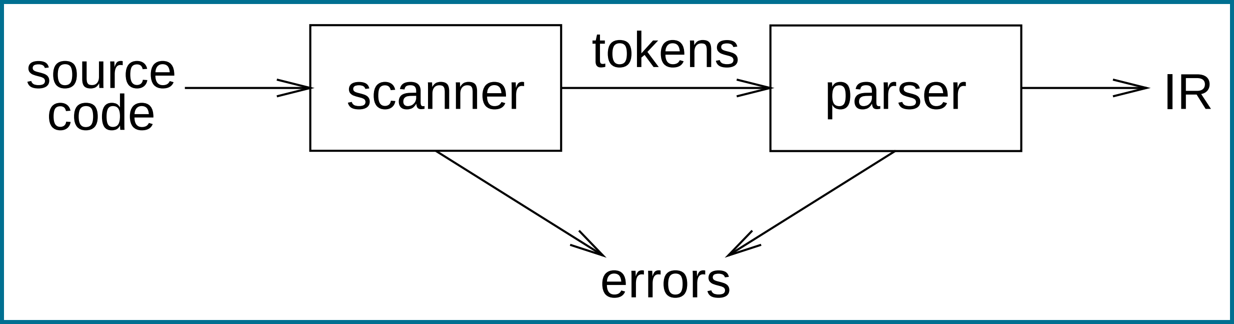


Can we build $n \times m$ compilers with $n + m$ components?

- must encode *all* the knowledge in each front end
- must represent *all* the features in one IR
- must handle *all* the features in each back end

Limited success with low-level IRs

Front end

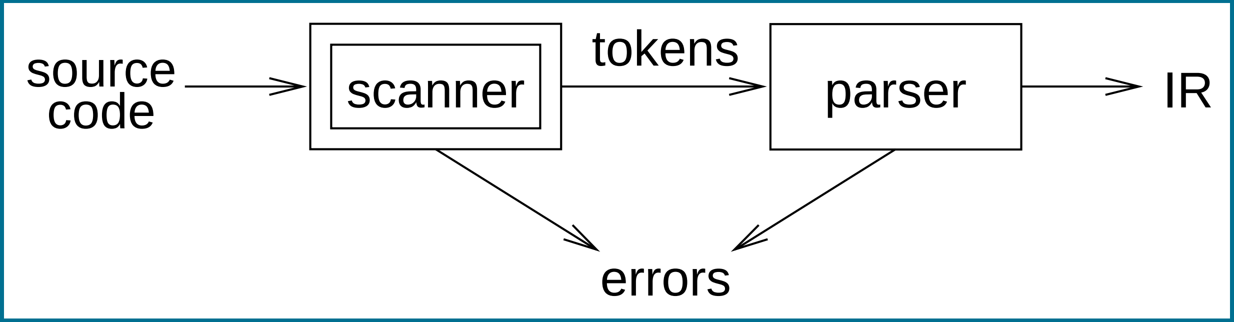


Responsibilities:

- recognize legal procedure
- report errors
- produce IR
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Front end



Scanner:

- maps characters into *tokens* – the basic unit of syntax

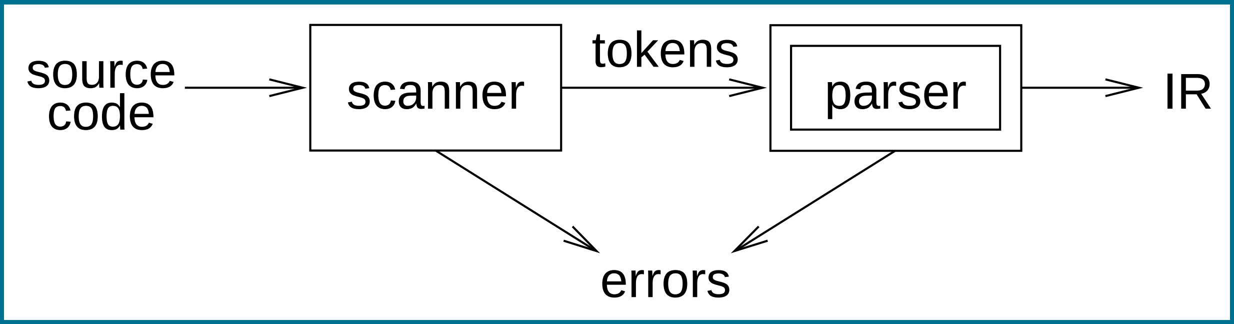
  `x = x + y;`

  becomes

  <id, `x`> = <id, `x`> + <id, `y`> ;
- character string value for a *token* is a *lexeme*
- typical tokens: *number*, *id*, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed

  $\Rightarrow$ use specialized recognizer (as opposed to `lex`)

Front end



Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

Parser generators mechanize much of the work

Front end

Context-free syntax is specified with a *grammar*

```
<sheep noise> ::= baa
               |  baa <sheep noise>
```

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

The format is called *Backus-Naur form* (BNF)

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

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

Front end

Context free syntax can be put to better use

1	<goal>	::=	<expr>
2	<expr>	::=	<expr> <op> <term>
3		|	<term>
4	<term>	::=	`number`
5		|	`id`
6	<op>	::=	`+`
7		|	`-`

This grammar defines simple expressions with addition and subtraction over the tokens `id` and `number`

S = *<goal>*

T = `number`, `id`, `+`, `-`

N = *<goal>*, *<expr>*, *<term>*, *<op>*

P = 1, 2, 3, 4, 5, 6, 7

Front end

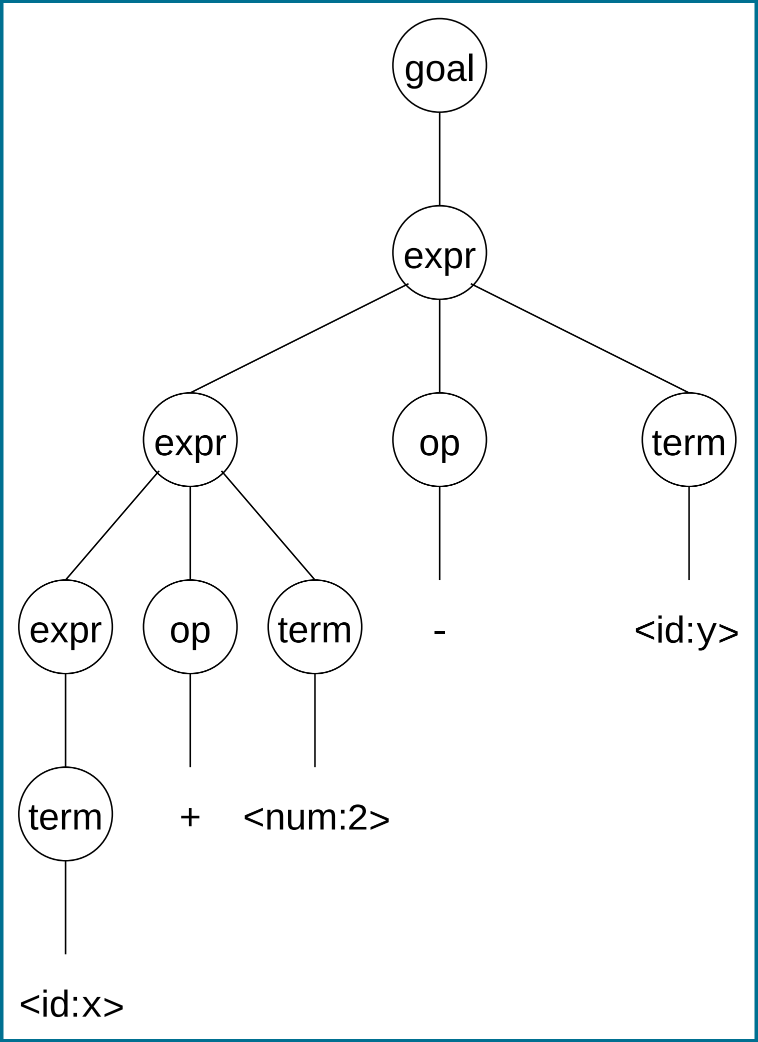
Given a grammar, valid sentences can be derived by repeated substitution.

Prod'n.	Result
	<goal>
1	<expr>
2	<expr> <op> <term>
5	<expr> <op> `y`
7	<expr> `- y`
2	<expr> <op> <term> `- y`
4	<expr> <op> `2 - y`
6	<expr> `+ 2 - y`
3	<term> `+ 2 - y`
5	`x + 2 - y`

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

Front end

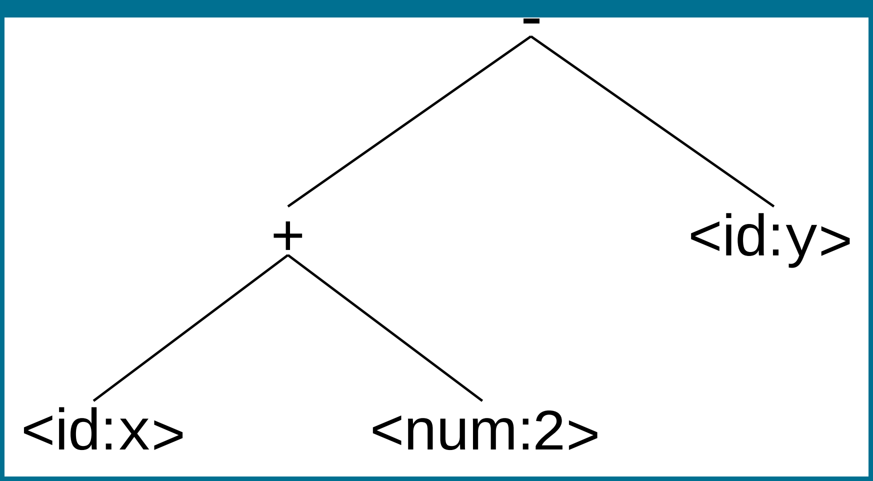
A parse can be represented by a tree called a *parse* or *syntax* tree



Obviously, this contains a lot of unnecessary information

Front end

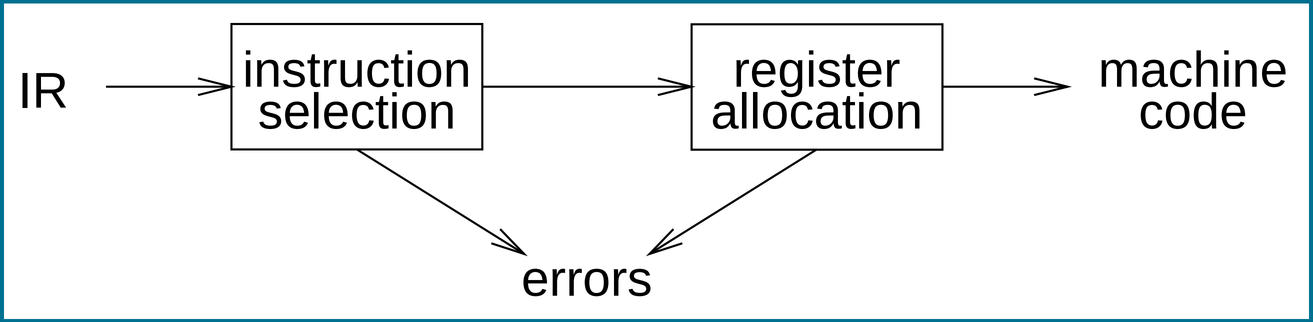
So, compilers often use an *abstract syntax tree*



This is much more concise

Abstract syntax trees (ASTs) are often used as an IR between front end and back end

Back end

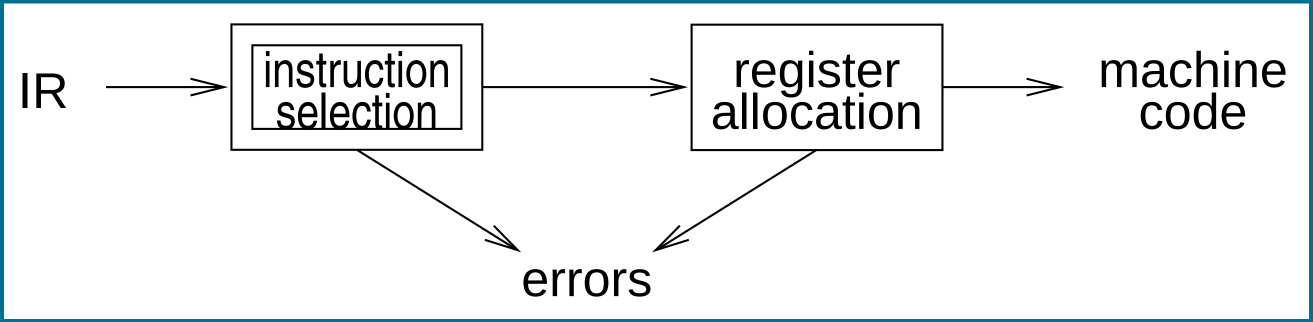


Responsibilities

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

Automation has been less successful here

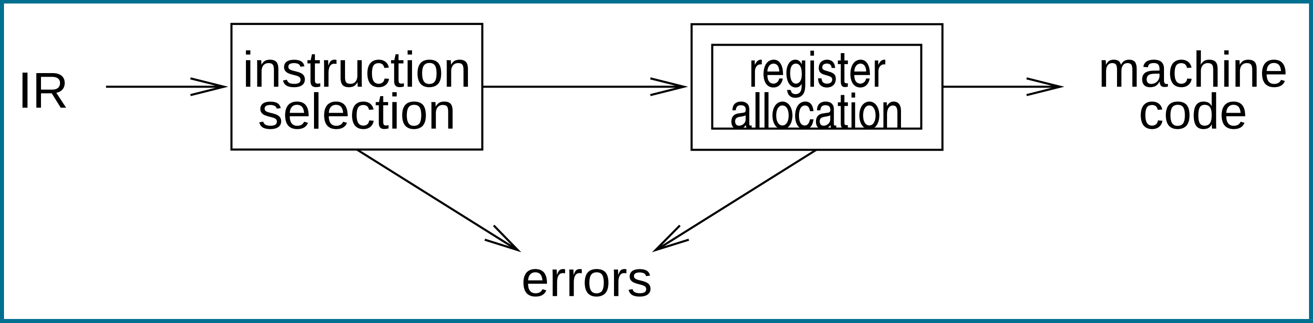
Back end



Instruction selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - *ad hoc* techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming

Back end

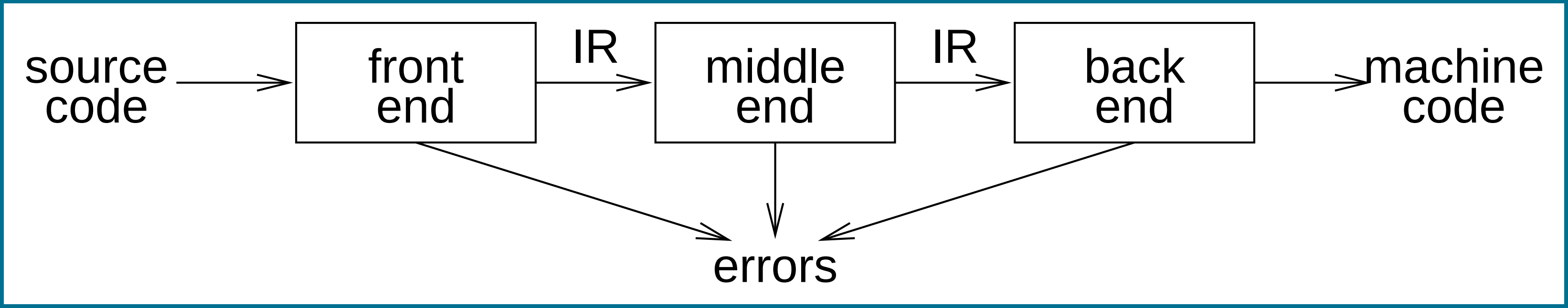


Register Allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

Modern allocators often use an analogy to graph coloring

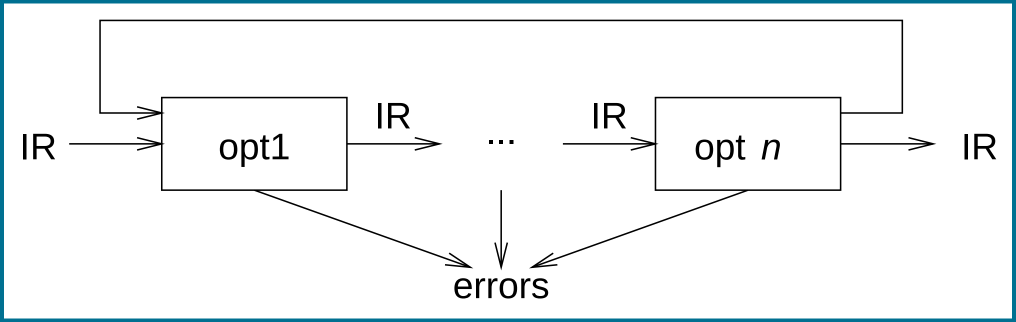
Traditional three pass compiler



Code Improvement

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values

Optimizer (middle end)

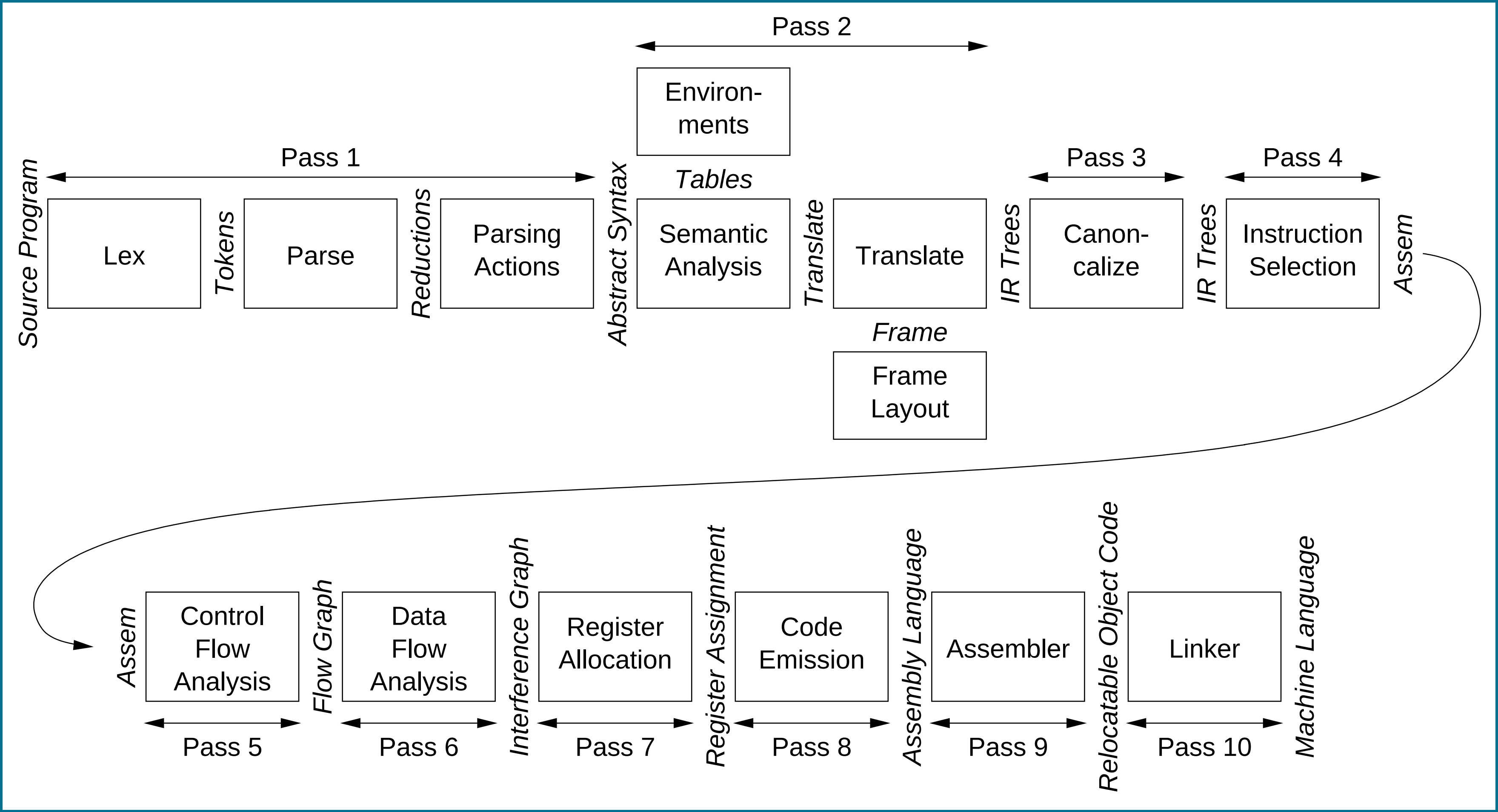


Modern optimizers are usually built as a set of passes

Typical passes

- constant propagation and folding
- code motion
- reduction of operator strength
- common subexpression elimination
- redundant store elimination
- dead code elimination

Compiler example



Compiler phases

Lex	Break source file into individual words, or *tokens*
Parse	Analyse the phrase structure of program
Parsing Actions	Build a piece of *abstract syntax tree* for each phrase
Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase
Frame Layout	Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way
Translate	Produce *intermediate representation trees* (IR trees), a notation that is not tied to any particular source language or target machine
Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases
Instruction Selection	Group IR-tree nodes into clumps that correspond to actions of target-machine instructions
Control Flow Analysis	Analyse sequence of instructions into *control flow graph* showing all possible flows of control program might follow when it runs
Data Flow Analysis	Gather information about flow of data through variables of program; e.g., *liveness analysis* calculates places where each variable holds a still-needed (*live*) value
Register Allocation	Choose registers for variables and temporary values; variables not simultaneously live can share same register
Code Emission	Replace temporary names in each machine instruction with registers

A straight-line programming language

- A straight-line programming language (no loops or conditionals):

Stm	$\rightarrow$	*Stm* ; *Stm*	CompoundStm
Stm	$\rightarrow$	`id` := *Exp*	AssignStm
Stm	$\rightarrow$	`print` (*ExpList*)	PrintStm
Exp	$\rightarrow$	`id`	IdExp
Exp	$\rightarrow$	`num`	NumExp
Exp	$\rightarrow$	*Exp Binop Exp*	OpExp
Exp	$\rightarrow$	(*Stm* , *Exp*)	EseqExp
ExpList	$\rightarrow$	*Exp* , *ExpList*	PairExpList
ExpList	$\rightarrow$	*Exp*	LastExpList
Binop	$\rightarrow$	$+$	Plus
Binop	$\rightarrow$	$-$	Minus
Binop	$\rightarrow$	$\times$	Times
Binop	$\rightarrow$	$/$	Div

- e.g.,

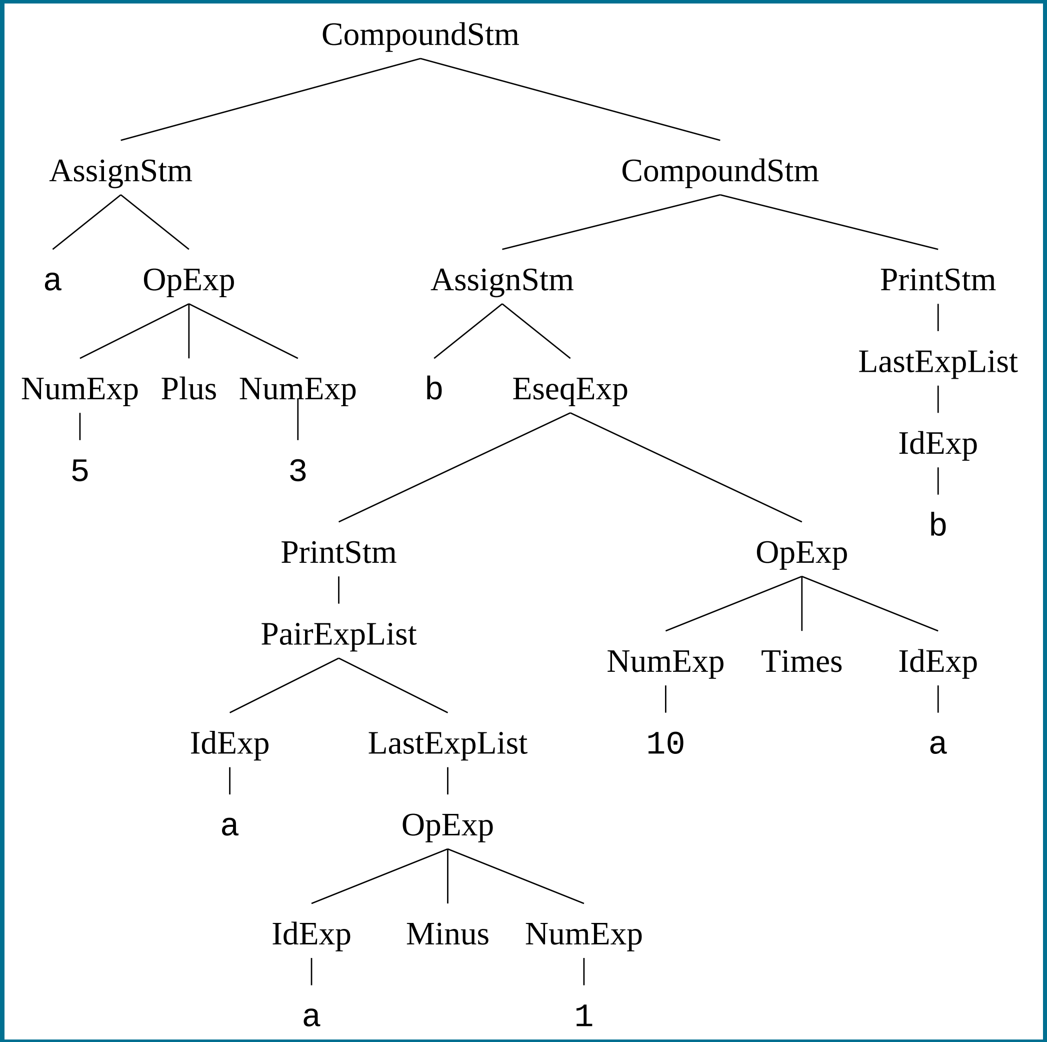
$$\mathtt{a} := 5+3;\ \mathtt{b} := (\mathtt{print}(\mathtt{a}, \mathtt{a}-1), 10 \times \mathtt{a});\ \mathtt{print}(\mathtt{b})$$

prints:

```
8 7
80
```

Tree representation

$$\mathtt{a} := 5+3;\ \mathtt{b} := (\mathtt{print}(\mathtt{a}, \mathtt{a}-1), 10 \times \mathtt{a});\ \mathtt{print}(\mathtt{b})$$



This is a convenient internal representation for a compiler to use.

Java classes for trees

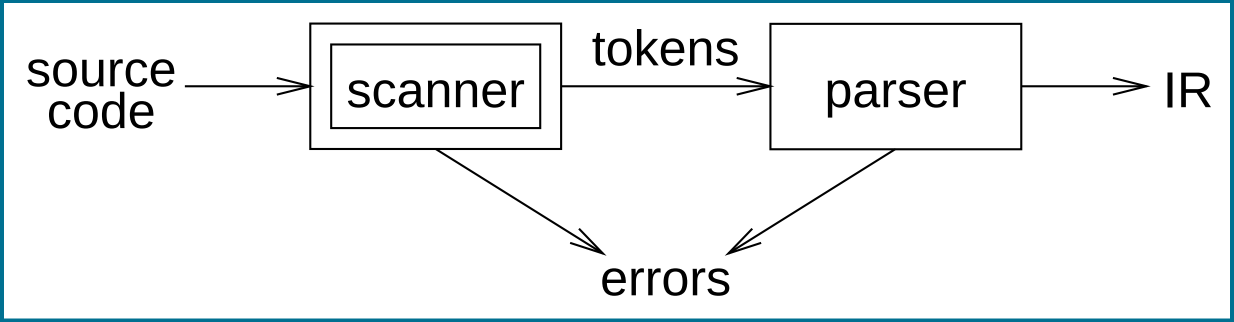
```
abstract class Stm {}
class CompoundStm extends Stm
   Stm stm1, stm2;
   CompoundStm(Stm s1, Stm s2)
   { stm1=s1; stm2=s2; }
}
class AssignStm extends Stm
{
   String id; Exp exp;
   AssignStm(String i, Exp e)
   { id=i; exp=e; }
}
class PrintStm extends Stm {
   ExpList exps;
   PrintStm(ExpList e)
   { exps=e; }
}

abstract class Exp {}
class IdExp extends Exp {
   String id;
   IdExp(String i) {id=i;}
}
```

```
class NumExp extends Exp {
   int num;
   NumExp(int n) {num=n;}
}
class OpExp extends Exp {
   Exp left, right; int oper;
   final static int
     Plus=1,Minus=2,Times=3,Div=4;
   OpExp(Exp l, int o, Exp r)
   { left=l; oper=o; right=r; }
}
class EseqExp extends Exp {
   Stm stm; Exp exp;
   EseqExp(Stm s, Exp e)
   { stm=s; exp=e; }
}
abstract class ExpList {}
class PairExpList extends ExpList {
   Exp head; ExpList tail;
   public PairExpList(Exp h, ExpList t)
   { head=h; tail=t; }
}
class LastExpList extends ExpList {
   Exp head;
   public LastExpList(Exp h) {head=h;}
}
```

Chapter 2: Lexical Analysis

Scanner



- maps characters into *tokens* – the basic unit of syntax
  `x = x + y;`
  becomes
  <id, `x`> = <id, `x`> + <id, `y`> `;`
- character string value for a *token* is a *lexeme*
- typical tokens: *number*, *id*, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed
  ⇒ use specialized recognizer (as opposed to `lex`)

Specifying patterns

A scanner must recognize various parts of the language's syntax

Some parts are easy:

white space

$$\begin{array}{lcl} \langle\text{ws}\rangle & ::= & \langle\text{ws}\rangle \texttt{' '} \\ & | & \langle\text{ws}\rangle \texttt{'\textbackslash t'} \\ & | & \texttt{' '} \\ & | & \texttt{'\textbackslash t'} \end{array}$$

keywords and operators

specified as literal patterns: `do`, `end`

comments

opening and closing delimiters: `/*` $\cdots$ `*/`

Specifying patterns

A scanner must recognize various parts of the language's syntax

Other parts are much harder:

identifi ers

alphabetic followed by k alphanumerics (_, $, &, . . .)

numbers

integers: 0 or digit from 1-9 followed by digits from 0-9

decimals: integer `'.'` digits from 0-9

reals: (integer or decimal) `'E'` (+ or -) digits from 0-9

complex: `'('` real `','` real `')'`

We need a powerful notation to specify these patterns

Operations on languages

Operation	Definition
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \in L$ or $s \in M\}$
concatenation of L and M written *LM*	$LM = \{st \mid s \in L$ and $t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular expressions

Patterns are often specified as *regular languages*

Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*

Regular expressions (*over an alphabet* Σ):

1. ε is a RE denoting the set $\{\varepsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$, then:

   (r) is a RE denoting $L(r)$

   $(r) \mid (s)$ is a RE denoting $L(r) \bigcup L(s)$

   $(r)(s)$ is a RE denoting $L(r)L(s)$

   $(r)^*$ is a RE denoting $L(r)^*$

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.

Examples

identifier

$\textit{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$

$\textit{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$\textit{id} \rightarrow \textit{letter} \; (\; \textit{letter} \mid \textit{digit} \;)^*$

numbers

$\textit{integer} \rightarrow (+ \mid - \mid \varepsilon) \; (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9) \; \textit{digit}^*)$

$\textit{decimal} \rightarrow \textit{integer} \; . \; (\; \textit{digit} \;)^*$

$\textit{real} \rightarrow (\; \textit{integer} \mid \textit{decimal} \;) \; \mathtt{E} \; (+ \mid -) \; \textit{digit}^*$

$\textit{complex} \rightarrow \texttt{'('} \; \textit{real} \; \texttt{,} \; \textit{real} \; \texttt{')'}$

Numbers can get much more complicated

Most programming language tokens can be described with REs

We can use REs to build scanners automatically

Algebraic properties of REs

Axiom	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\varepsilon r = r$ $r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

Examples

Let $\Sigma = \{a, b\}$

1. $a|b$ denotes $\{a, b\}$

2. $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
   i.e., $(a|b)(a|b) = aa|ab|ba|bb$

3. a^* denotes $\{\varepsilon, a, aa, aaa, \ldots\}$

4. $(a|b)^*$ denotes the set of all strings of a's and b's (including ε)
   i.e., $(a|b)^* = (a^*b^*)^*$

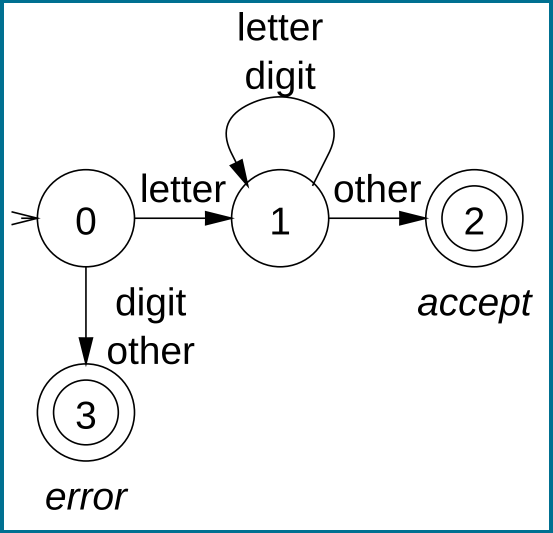
5. $a|a^*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Recognizers

From a regular expression we can construct a

deterministic finite automaton (DFA)

Recognizer for *identifier*:



identifier

letter $\rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$

digit $\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

id $\rightarrow$ *letter* (*letter* | *digit*)*

Code for the recognizer

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:     /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:     /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:     /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Tables for the recognizer

Two tables control the recognizer

`char_class:`

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`next_state:`

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

To change languages, we can just change tables

Automatic construction

Scanner generators automatically construct code from regular expression-like descriptions

- construct a *dfa*
- use state minimization techniques
- emit code for the scanner

  (table driven or direct code)

A key issue in automation is an interface to the parser

`lex` is a scanner generator supplied with UNIX

- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)

Grammars for regular languages

Can we place a restriction on the *form* of a grammar to ensure that it describes a regular language?

Provable fact:

For any RE r, there is a grammar g such that $L(r) = L(g)$.

The grammars that generate regular sets are called *regular grammars*

Definition:

In a regular grammar, all productions have one of two forms:

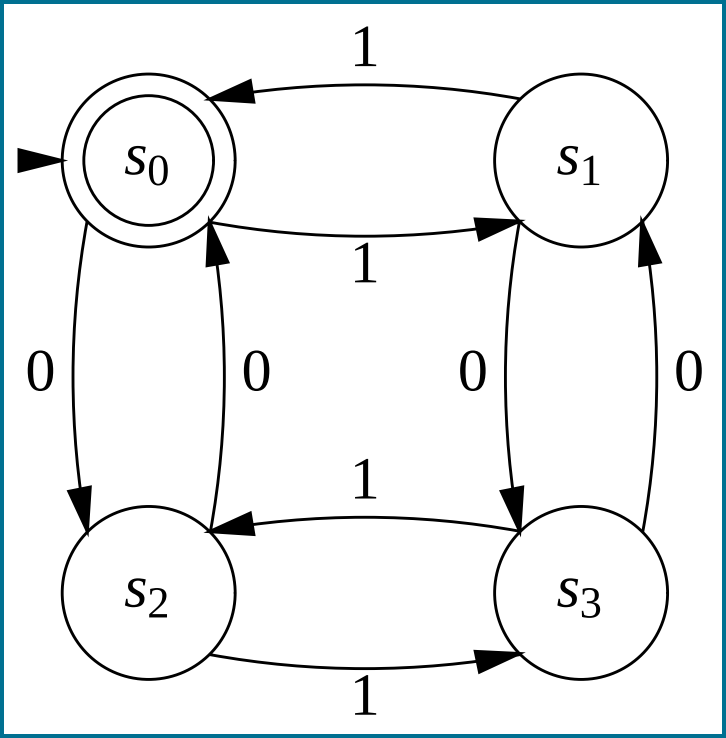
1. $A \rightarrow aA$
2. $A \rightarrow a$

where A is any non-terminal and a is any terminal symbol

These are also called *type 3* grammars (Chomsky)

More regular languages

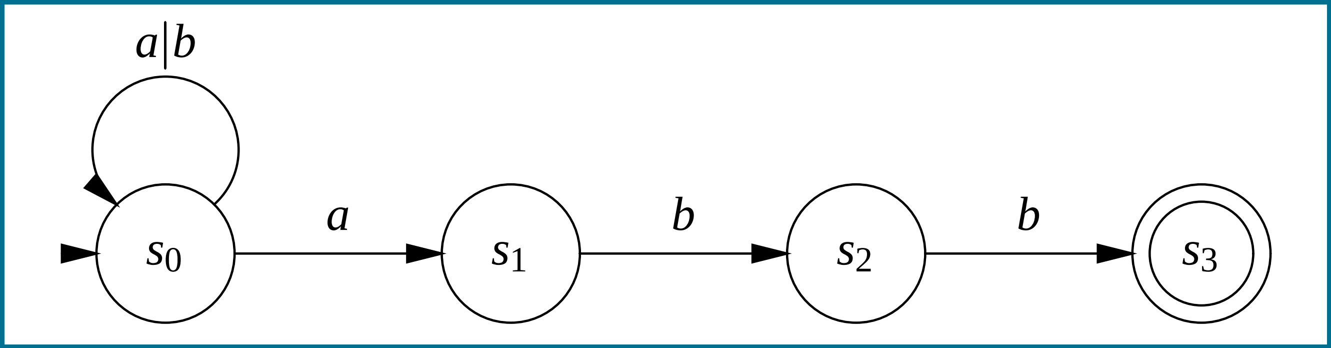
Example: the set of strings containing an even number of zeros and an even number of ones



The RE is $(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$

More regular expressions

What about the RE $(a \mid b)^*abb$?



State s_0 has multiple transitions on a!

$\Rightarrow$ *nondeterministic finite automaton*

	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	$-$	$\{s_2\}$
s_2	$-$	$\{s_3\}$

Finite automata

A *non-deterministic finite automaton* (NFA) consists of:

1. a set of *states* $S = \{s_0, \ldots, s_n\}$
2. a set of input symbols Σ (the alphabet)
3. a transition function *move* mapping state-symbol pairs to sets of states
4. a distinguished *start state* s_0
5. a set of distinguished *accepting* or *final* states F

A *Deterministic Finite Automaton* (DFA) is a special case of an NFA:

1. no state has a ε-transition, and
2. for each state s and input symbol a, there is at most one edge labelled a leaving s.

A DFA accepts x iff. there exists a *unique* path through the transition graph from the s_0 to an accepting state such that the labels along the edges spell x.

DFAs and NFAs are equivalent

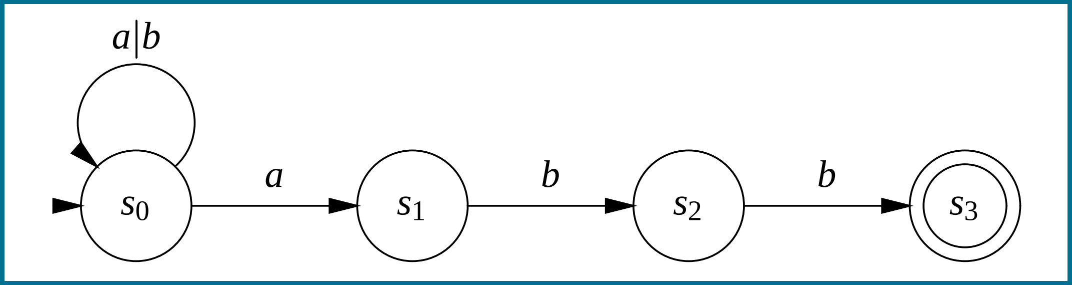
1. DFAs are clearly a subset of NFAs

2. Any NFA can be converted into a DFA, by simulating sets of simultaneous states:

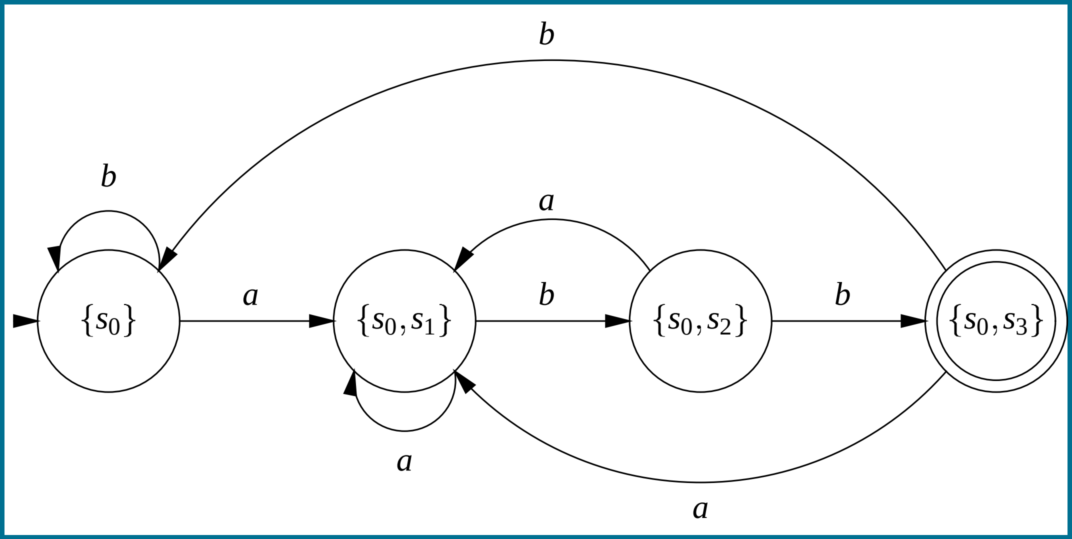
   - each DFA state corresponds to a set of NFA states

   - possible exponential blowup

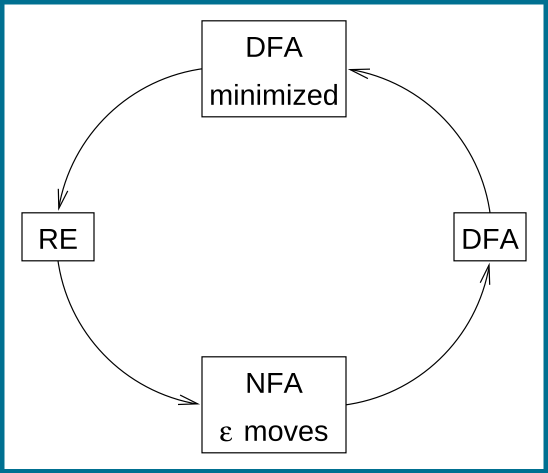
NFA to DFA using the subset construction: example 1



	a	b
$\{s_0\}$	$\{s_0, s_1\}$	$\{s_0\}$
$\{s_0, s_1\}$	$\{s_0, s_1\}$	$\{s_0, s_2\}$
$\{s_0, s_2\}$	$\{s_0, s_1\}$	$\{s_0, s_3\}$
$\{s_0, s_3\}$	$\{s_0, s_1\}$	$\{s_0\}$



Constructing a DFA from a regular expression



RE $\rightarrow$NFA w/ε moves
- build NFA for each term
- connect them with ε moves

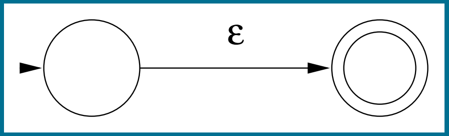
NFA w/ε moves to DFA
- construct the simulation
- the “subset” construction

DFA $\rightarrow$ minimized DFA
- merge compatible states

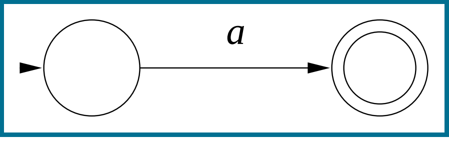
DFA $\rightarrow$ RE
- construct $R^k_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$

RE to NFA

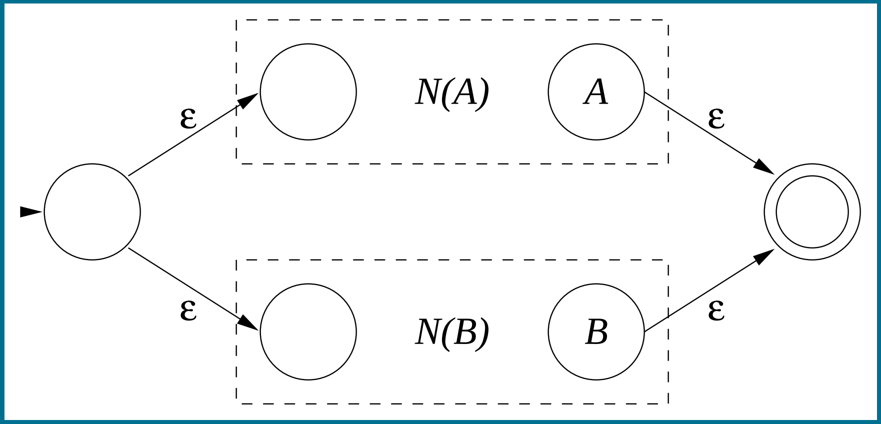
$N(\varepsilon)$



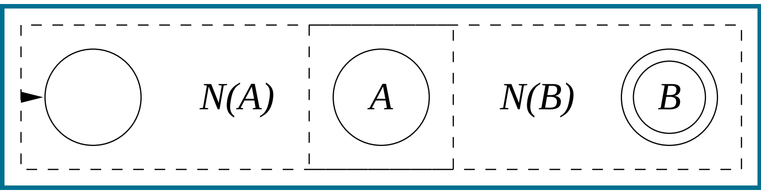
$N(a)$



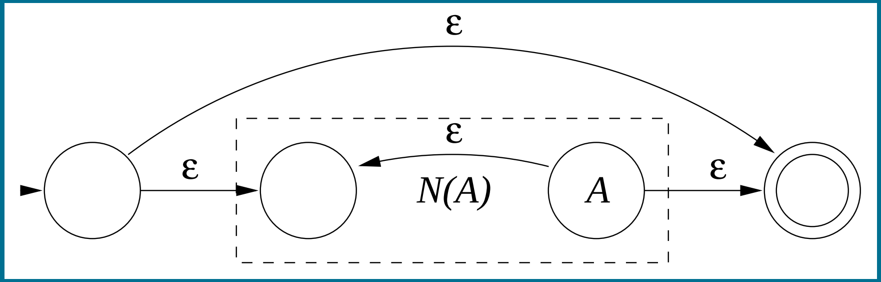
$N(A|B)$



$N(AB)$

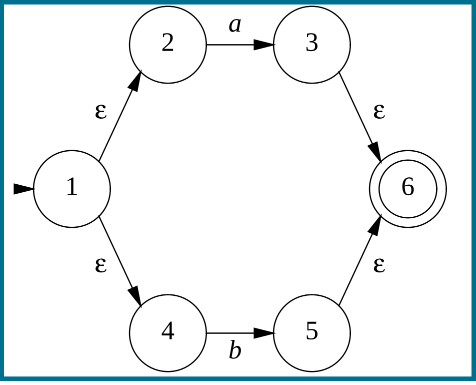


$N(A^*)$

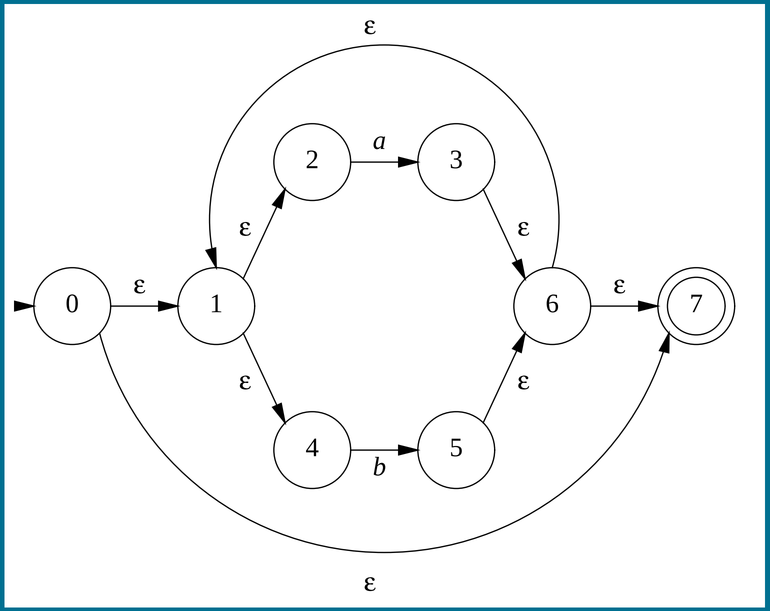


RE to NFA: example

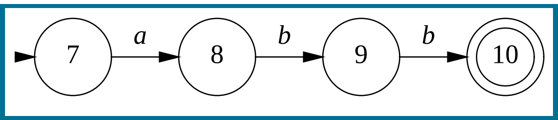
$(a \mid b)^{*}abb$



$a|b$



$(a|b)^{*}$



abb

NFA to DFA: the subset construction

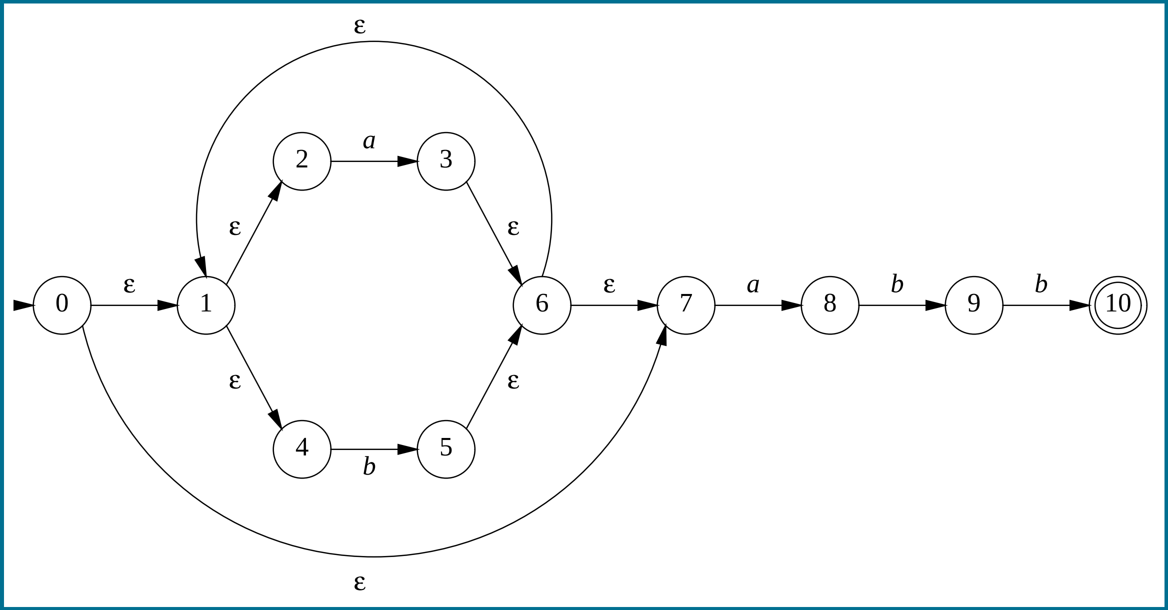
Input:	NFA N
Output:	A DFA D with states *Dstates* and transitions *Dtrans* such that $L(D) = L(N)$
Method:	Let s be a state in N and T be a set of states, and using the following operations:

Operation	Defi nition
ε-*closure*(s)	set of NFA states reachable from NFA state s on ε-transitions alone
ε-*closure*(T)	set of NFA states reachable from some NFA state s in T on ε-transitions alone
move(T, a)	set of NFA states to which there is a transition on input symbol a from some NFA state s in T

```
add state $T = \varepsilon$-closure$(s_0)$ unmarked to Dstates
while $\exists$ unmarked state $T$ in Dstates
    mark $T$
    for each input symbol $a$
        $U = \varepsilon$-closure(move$(T, a)$)
        if $U \notin$ Dstates then add $U$ to Dstates unmarked
        Dtrans$[T, a] = U$
    endfor
endwhile
```

ε-*closure*(s_0) is the start state of D
A state of D is accepting if it contains at least one accepting state in N

NFA to DFA using subset construction: example 2



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

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

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

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

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

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Limits of regular languages

Not all languages are regular

One cannot construct DFAs to recognize these languages:

- $L = \{p^k q^k\}$
- $L = \{wcw^r \mid w \in \Sigma^*\}$

Note: neither of these is a regular expression!
(DFAs cannot count!)

But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
  $(\varepsilon \mid 1)(01)^*(\varepsilon \mid 0)$
- sets of pairs of 0's and 1's
  $(01 \mid 10)^+$

So what is hard?

Language features that can cause problems:

reserved words

PL/I had no reserved words

`if then then then = else; else else = then;`

signifi cant blanks

FORTRAN and Algol68 ignore blanks

`do 10 i = 1,25`

`do 10 i = 1.25`

string constants

special characters in strings

`newline`, `tab`, `quote`, `comment delimiter`

fi nite closures

some languages limit identifier lengths

adds states to count length

FORTRAN 66 $\rightarrow$ 6 characters

These can be swept under the rug in the language design

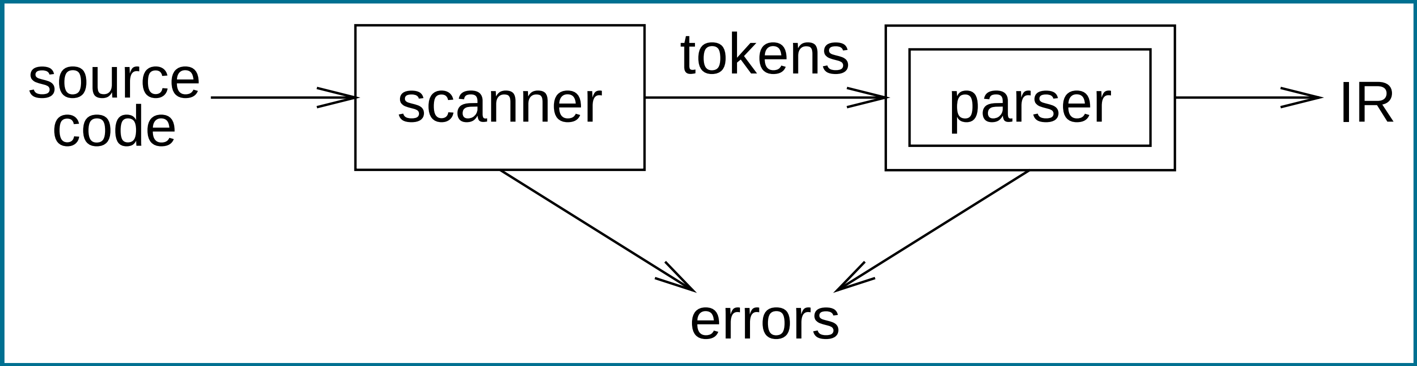
How bad can it get?

```
      INTEGERFUNCTIONA
      PARAMETER(A=6,B=2)
      IMPLICIT CHARACTER*(A-B)(A-B)
      INTEGER FORMAT(10),IF(10),DO9E1
100   FORMAT(4H)=(3)
200   FORMAT(4 )=(3)
      DO9E1=1
      DO9E1=1,2
         IF(X)=1
         IF(X)H=1
         IF(X)300,200
300      CONTINUE
      END
C     this is a comment
    $ FILE(1)
      END
```

Example due to Dr. F.K. Zadeck of IBM Corporation

Chapter 3: LL Parsing

The role of the parser



Parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

For the next few weeks, we will look at parser construction

Syntax analysis

Context-free syntax is specified with a *context-free grammar*.

Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

V_t is the set of *terminal* symbols in the grammar.
For our purposes, V_t is the set of tokens returned by the scanner.

V_n, the *nonterminals*, is a set of syntactic variables that denote sets of (sub)strings occurring in the language.
These are used to impose a structure on the grammar.

S is a distinguished nonterminal $(S \in V_n)$ denoting the entire set of strings in $L(G)$.
This is sometimes called a *goal symbol*.

P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language.
Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the *vocabulary* of G

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

If $A \to \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \to \gamma$

Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps

If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G

$L(G) = \{w \in V_t^* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a *sentence* of G

Note, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).

Example:

$$\begin{array}{r|rcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
3 & & \mid & \texttt{num} \\
4 & & \mid & \texttt{id} \\
5 & \langle\text{op}\rangle & ::= & + \\
6 & & \mid & - \\
7 & & \mid & * \\
8 & & \mid & /
\end{array}$$

This describes simple expressions over numbers and identifiers.

In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or <u>underline</u>
3. productions as in the example

Scanning vs. parsing

Where do we draw the line?

$$\begin{array}{lcl} \textit{term} & ::= & [\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}]([\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}] \mid [0-9])^* \\ & \mid & \mathtt{0} \mid [\mathtt{1}-\mathtt{9}][\mathtt{0}-\mathtt{9}]^* \\ \textit{op} & ::= & + \mid - \mid * \mid / \\ \textit{expr} & ::= & (\textit{term op})^* \textit{term} \end{array}$$

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

Context-free grammars are used to count:

- brackets: `()`, `begin`...`end`, `if`...`then`...`else`
- imparting structure: expressions

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes compiler more manageable.

Derivations

We can view the productions of a CFG as rewriting rules.

Using our example CFG:

$$\begin{array}{rcl}
\langle \text{goal} \rangle & \Rightarrow & \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{id,x} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{id,x} \rangle + \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{id,x} \rangle + \langle \text{num,2} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{id,x} \rangle + \langle \text{num,2} \rangle * \langle \text{expr} \rangle \\
& \Rightarrow & \langle \text{id,x} \rangle + \langle \text{num,2} \rangle * \langle \text{id,y} \rangle
\end{array}$$

We have derived the sentence $\texttt{x} + \texttt{2} * \texttt{y}$.
We denote this $\langle \text{goal} \rangle \Rightarrow^* \texttt{id} + \texttt{num} * \texttt{id}$.

Such a sequence of rewrites is a *derivation* or a *parse*.

The process of discovering a derivation is called *parsing*.

Derivations

At each step, we chose a non-terminal to replace.

This choice can lead to different derivations.

Two are of particular interest:

leftmost derivation
the leftmost non-terminal is replaced at each step

rightmost derivation
the rightmost non-terminal is replaced at each step

The previous example was a leftmost derivation.

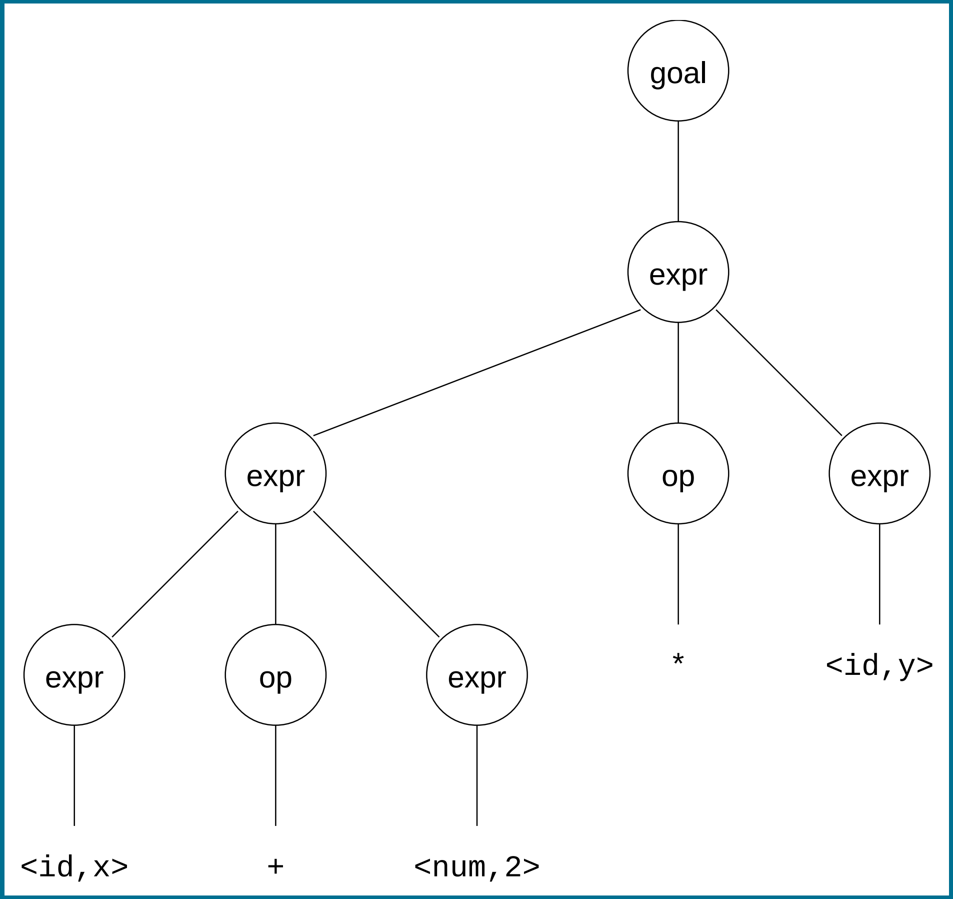
Rightmost derivation

For the string $\mathtt{x} + 2 * \mathtt{y}$:

$$\begin{array}{rcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{id,y}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle * \langle\text{id,y}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle * \langle\text{id,y}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
 & \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
 & \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{array}$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.

Precedence



Treewalk evaluation computes $(\mathtt{x} + 2) * \mathtt{y}$
— the “wrong” answer!

Should be $\mathtt{x} + (2 * \mathtt{y})$

Precedence

These two derivations point out a problem with the grammar.

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

To add precedence takes additional machinery:

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	`num`
9		$\mid$	`id`

This grammar enforces a precedence on the derivation:

- terms *must* be derived from expressions
- forces the "correct" tree

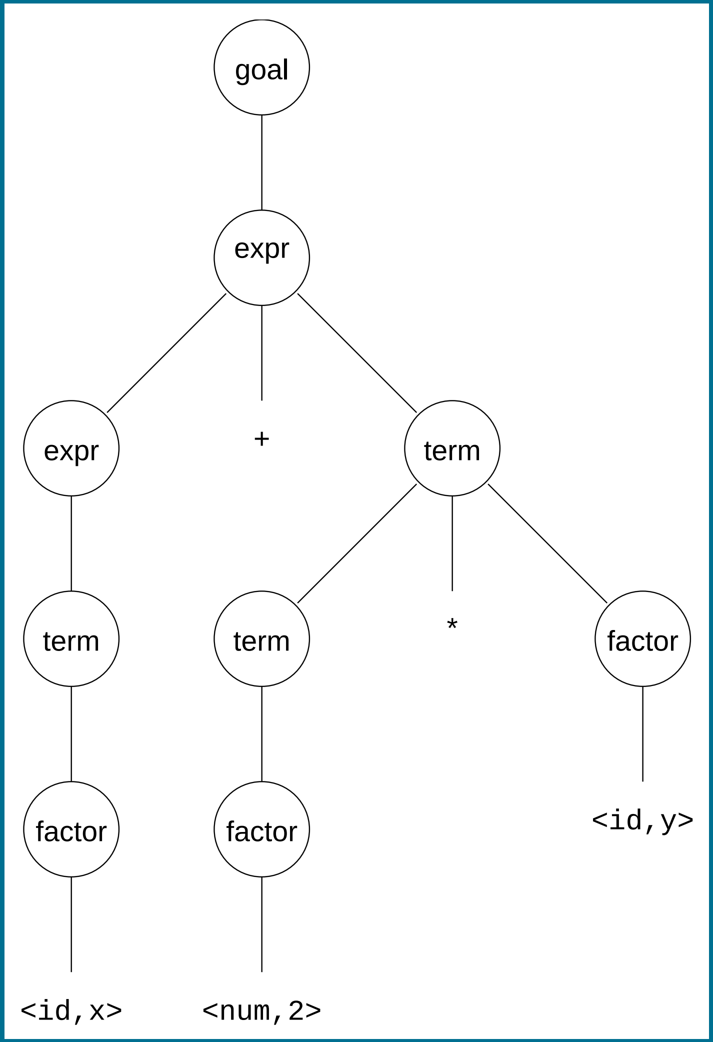
Precedence

Now, for the string $\mathtt{x} + \mathtt{2} * \mathtt{y}$:

$$\begin{array}{rcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{id,}\mathtt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{factor}\rangle * \langle\text{id,}\mathtt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,}\mathtt{2}\rangle * \langle\text{id,}\mathtt{y}\rangle \\
& \Rightarrow & \langle\text{term}\rangle + \langle\text{num,}\mathtt{2}\rangle * \langle\text{id,}\mathtt{y}\rangle \\
& \Rightarrow & \langle\text{factor}\rangle + \langle\text{num,}\mathtt{2}\rangle * \langle\text{id,}\mathtt{y}\rangle \\
& \Rightarrow & \langle\text{id,}\mathtt{x}\rangle + \langle\text{num,}\mathtt{2}\rangle * \langle\text{id,}\mathtt{y}\rangle
\end{array}$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$, but this time, we build the desired tree.

Precedence



Treewalk evaluation computes $\mathtt{x} + (2 * \mathtt{y})$

Ambiguity

If a grammar has more than one derivation for a single sentential form, then it is *ambiguous*

Example:

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{then } \langle \text{stmt} \rangle \\ & \mid & \texttt{if } \langle \text{expr} \rangle \texttt{then } \langle \text{stmt} \rangle \texttt{else } \langle \text{stmt} \rangle \\ & \mid & \texttt{other stmts} \end{array}$$

Consider deriving the sentential form:

$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

$$\begin{array}{lcl}
\langle\text{stmt}\rangle & ::= & \langle\text{matched}\rangle \\
 & \mid & \langle\text{unmatched}\rangle \\
\langle\text{matched}\rangle & ::= & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{matched}\rangle\ \texttt{else}\ \langle\text{matched}\rangle \\
 & \mid & \texttt{other stmts} \\
\langle\text{unmatched}\rangle & ::= & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle \\
 & \mid & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{matched}\rangle\ \texttt{else}\ \langle\text{unmatched}\rangle
\end{array}$$

This generates the same language as the ambiguous grammar, but applies the common sense rule:

match each `else` *with the closest unmatched* `then`

This is most likely the language designer’s intent.

Ambiguity

Ambiguity is often due to confusion in the context-free specification.

Context-sensitive confusions can arise from *overloading*.

Example:

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

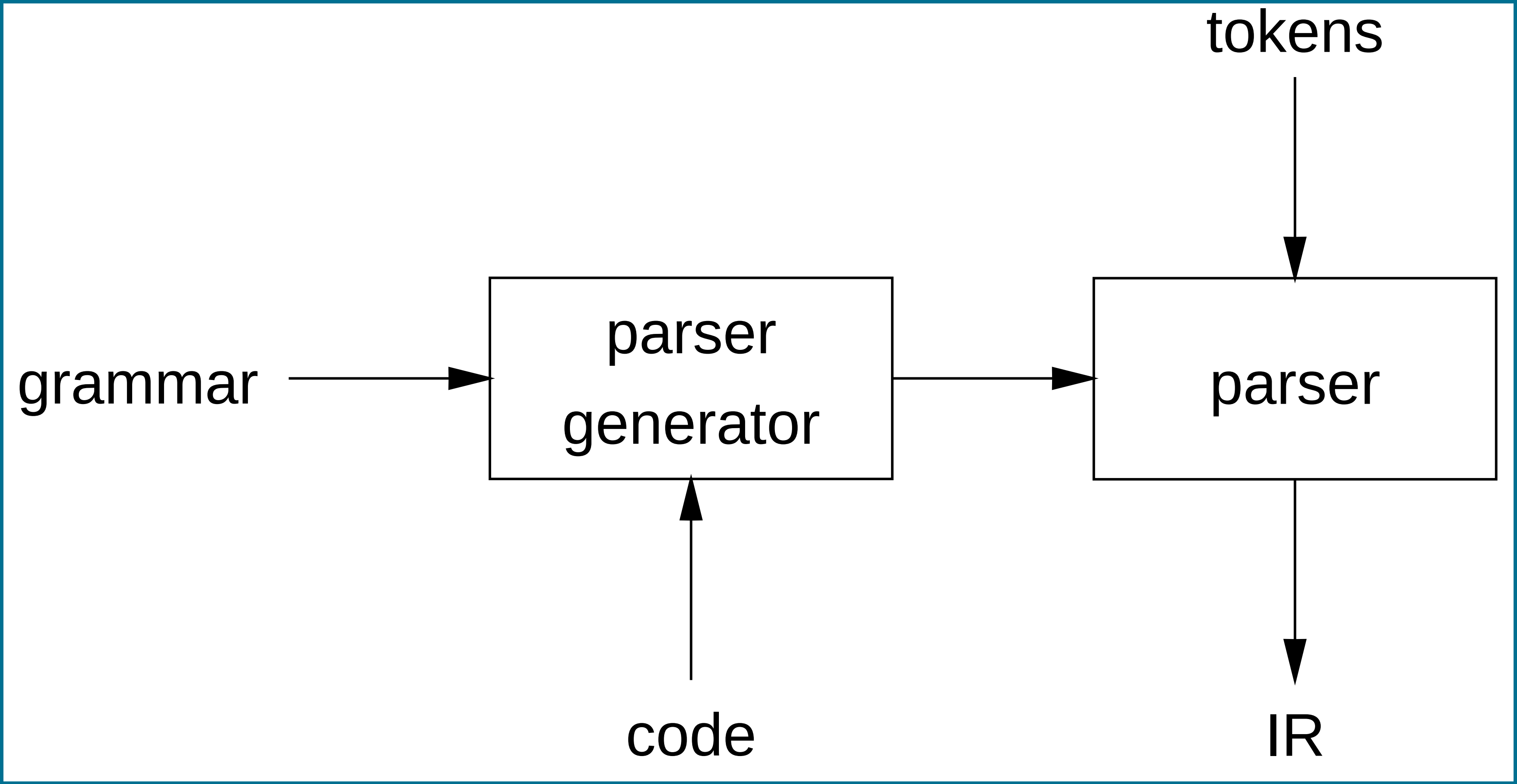
In many Algol-like languages, `f` could be a function or subscripted variable.

Disambiguating this statement requires context:

- need *values* of declarations
- not *context-free*
- really an issue of *type*

Rather than complicate parsing, we will handle this separately.

Parsing: the big picture



Our goal is a flexible parser generator system

Top-down versus bottom-up

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid prefi xes*)
- use a stack to store both state and sentential forms

Top-down parsing

A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar.

To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find the next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1

$\Rightarrow$ should be guided by input string

Simple expression grammar

Recall our grammar for simple expressions:

$$
\begin{array}{r|lcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
3 & & | & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
4 & & | & \langle\text{term}\rangle \\
5 & \langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
6 & & | & \langle\text{term}\rangle / \langle\text{factor}\rangle \\
7 & & | & \langle\text{factor}\rangle \\
8 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
9 & & | & \texttt{id}
\end{array}
$$

Consider the input string $\texttt{x} - \texttt{2} * \texttt{y}$

Example

Prod’n	Sentential form	Input
–	⟨goal⟩	↑x – 2 ∗ y
1	⟨expr⟩	↑x – 2 ∗ y
2	⟨expr⟩ + ⟨term⟩	↑x – 2 ∗ y
4	⟨term⟩ + ⟨term⟩	↑x – 2 ∗ y
7	⟨factor⟩ + ⟨term⟩	↑x – 2 ∗ y
9	id + ⟨term⟩	↑x – 2 ∗ y
–	id + ⟨term⟩	x ↑– 2 ∗ y
–	⟨expr⟩	↑x – 2 ∗ y
3	⟨expr⟩ – ⟨term⟩	↑x – 2 ∗ y
4	⟨term⟩ – ⟨term⟩	↑x – 2 ∗ y
7	⟨factor⟩ – ⟨term⟩	↑x – 2 ∗ y
9	id – ⟨term⟩	↑x – 2 ∗ y
–	id – ⟨term⟩	x ↑– 2 ∗ y
–	id – ⟨term⟩	x – ↑2 ∗ y
7	id – ⟨factor⟩	x – ↑2 ∗ y
8	id – num	x – ↑2 ∗ y
–	id – num	x – 2 ↑∗ y
–	id – ⟨term⟩	x – ↑2 ∗ y
5	id – ⟨term⟩ ∗ ⟨factor⟩	x – ↑2 ∗ y
7	id – ⟨factor⟩ ∗ ⟨factor⟩	x – ↑2 ∗ y
8	id – num ∗ ⟨factor⟩	x – ↑2 ∗ y
–	id – num ∗ ⟨factor⟩	x – 2 ↑∗ y
–	id – num ∗ ⟨factor⟩	x – 2 ∗ ↑y
9	id – num ∗ id	x – 2 ∗ ↑y
–	id – num ∗ id	x – 2 ∗ y ↑

Example

Another possible parse for $\texttt{x} - \texttt{2} * \texttt{y}$

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$
2	$\cdots$	$\uparrow\texttt{x} \; - \; \texttt{2} \; * \; \texttt{y}$

If the parser makes the wrong choices, expansion doesn't terminate.
This isn't a good property for a parser to have.

(Parsers should terminate!)

Left-recursion

Top-down parsers cannot handle left-recursion in a grammar

Formally, a grammar is *left-recursive* if

$\exists A \in V_n$ such that $A \Rightarrow^+ A\alpha$ for some string α

Our simple expression grammar is left-recursive

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

Consider the grammar fragment:

$$\begin{array}{rcl} \langle\text{foo}\rangle & ::= & \langle\text{foo}\rangle\alpha \\ & | & \beta \end{array}$$

where α and β do not start with $\langle\text{foo}\rangle$

We can rewrite this as:

$$\begin{array}{rcl} \langle\text{foo}\rangle & ::= & \beta\langle\text{bar}\rangle \\ \langle\text{bar}\rangle & ::= & \alpha\langle\text{bar}\rangle \\ & | & \varepsilon \end{array}$$

where $\langle\text{bar}\rangle$ is a new non-terminal

This fragment contains no left-recursion

Example

Our expression grammar contains two cases of left-recursion

$$\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
 & | & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
 & | & \langle\text{term}\rangle \\
\langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
 & | & \langle\text{term}\rangle / \langle\text{factor}\rangle \\
 & | & \langle\text{factor}\rangle
\end{array}$$

Applying the transformation gives

$$\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\langle\text{expr}'\rangle \\
\langle\text{expr}'\rangle & ::= & +\langle\text{term}\rangle\langle\text{expr}'\rangle \\
 & | & \varepsilon \\
 & | & -\langle\text{term}\rangle\langle\text{expr}'\rangle \\
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\langle\text{term}'\rangle \\
\langle\text{term}'\rangle & ::= & *\langle\text{factor}\rangle\langle\text{term}'\rangle \\
 & | & \varepsilon \\
 & | & /\langle\text{factor}\rangle\langle\text{term}'\rangle
\end{array}$$

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{term}\rangle + \langle\text{expr}\rangle$
3		$\mid$	$\langle\text{term}\rangle - \langle\text{expr}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{factor}\rangle * \langle\text{term}\rangle$
6		$\mid$	$\langle\text{factor}\rangle / \langle\text{term}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	`num`
9		$\mid$	`id`

It is

- right-recursive
- free of ε productions

Unfortunately, it generates different associativity
Same syntax, different meaning

Example

Our long-suffering expression grammar:

$$
\begin{array}{r|lcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\langle\text{expr}'\rangle \\
3 & \langle\text{expr}'\rangle & ::= & +\langle\text{term}\rangle\langle\text{expr}'\rangle \\
4 & & | & -\langle\text{term}\rangle\langle\text{expr}'\rangle \\
5 & & | & \varepsilon \\
6 & \langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\langle\text{term}'\rangle \\
7 & \langle\text{term}'\rangle & ::= & *\langle\text{factor}\rangle\langle\text{term}'\rangle \\
8 & & | & /\langle\text{factor}\rangle\langle\text{term}'\rangle \\
9 & & | & \varepsilon \\
10 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
11 & & | & \texttt{id}
\end{array}
$$

Recall, we factored out left-recursion

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

  Aho, Hopcroft, and Ullman, Problem 2.34

  Parsing, Translation and Compiling, Chapter 4

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are:

LL(1): **l**eft to right scan, **l**eft-most derivation, **1**-token lookahead; and
LR(1): **l**eft to right scan, **r**ight-most derivation, **1**-token lookahead

Predictive parsing

Basic idea:

> For any two productions $A \rightarrow \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For some RHS $\alpha \in G$, define FIRST(α) as the set of tokens that appear first in some string derived from α
That is, for some $w \in V_t^*$, $w \in \text{FIRST}(\alpha)$ iff. $\alpha \Rightarrow^* w\gamma$.

Key property:
Whenever two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

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

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example grammar has this property!

Left factoring

What if a grammar does not have this property?

Sometimes, we can transform a grammar to have this property.

For each non-terminal A find the longest prefix
α common to two or more of its alternatives.

if $\alpha \neq \varepsilon$ then replace all of the A productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$
with

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where A' is a new non-terminal.

Repeat until no two alternatives for a single
non-terminal have a common prefix.

Example

Consider a *right-recursive* version of the expression grammar:

$$\begin{array}{r|lcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{term}\rangle + \langle\text{expr}\rangle \\
3 & & | & \langle\text{term}\rangle - \langle\text{expr}\rangle \\
4 & & | & \langle\text{term}\rangle \\
5 & \langle\text{term}\rangle & ::= & \langle\text{factor}\rangle * \langle\text{term}\rangle \\
6 & & | & \langle\text{factor}\rangle / \langle\text{term}\rangle \\
7 & & | & \langle\text{factor}\rangle \\
8 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
9 & & | & \texttt{id}
\end{array}$$

To choose between productions 2, 3, & 4, the parser must see past the `num` or `id` and look at the $+$, $-$, $*$, or $/$.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \phi$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals that must be left factored:

$$\begin{array}{rcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle + \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle - \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle * \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle / \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle
\end{array}$$

Applying the transformation gives us:

$$\begin{array}{rcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & +\langle \text{expr} \rangle \\
& | & -\langle \text{expr} \rangle \\
& | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & *\langle \text{term} \rangle \\
& | & /\langle \text{term} \rangle \\
& | & \varepsilon
\end{array}$$

Example

Substituting back into the grammar yields

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle \langle \text{expr}' \rangle$
3	$\langle \text{expr}' \rangle$	$::=$	$+\langle \text{expr} \rangle$
4		$\mid$	$-\langle \text{expr} \rangle$
5		$\mid$	ε
6	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
7	$\langle \text{term}' \rangle$	$::=$	$*\langle \text{term} \rangle$
8		$\mid$	$/\langle \text{term} \rangle$
9		$\mid$	ε
10	$\langle \text{factor} \rangle$	$::=$	`num`
11		$\mid$	`id`

Now, selection requires only a single token lookahead.

Note: *This grammar is still right-associative.*

Example

	Sentential form	Input
–	⟨goal⟩	↑x – 2 ∗ y
1	⟨expr⟩	↑x – 2 ∗ y
2	⟨term⟩⟨expr′⟩	↑x – 2 ∗ y
6	⟨factor⟩⟨term′⟩⟨expr′⟩	↑x – 2 ∗ y
11	id⟨term′⟩⟨expr′⟩	↑x – 2 ∗ y
–	id⟨term′⟩⟨expr′⟩	x ↑- 2 ∗ y
9	idε ⟨expr′⟩	x ↑- 2
4	id– ⟨expr⟩	x ↑- 2 ∗ y
–	id– ⟨expr⟩	x – ↑2 ∗ y
2	id– ⟨term⟩⟨expr′⟩	x – ↑2 ∗ y
6	id– ⟨factor⟩⟨term′⟩⟨expr′⟩	x – ↑2 ∗ y
10	id– num⟨term′⟩⟨expr′⟩	x – ↑2 ∗ y
–	id– num⟨term′⟩⟨expr′⟩	x – 2 ↑* y
7	id– num∗ ⟨term⟩⟨expr′⟩	x – 2 ↑* y
–	id– num∗ ⟨term⟩⟨expr′⟩	x – 2 ∗ ↑y
6	id– num∗ ⟨factor⟩⟨term′⟩⟨expr′⟩	x – 2 ∗ ↑y
11	id– num∗ id⟨term′⟩⟨expr′⟩	x – 2 ∗ ↑y
–	id– num∗ id⟨term′⟩⟨expr′⟩	x – 2 ∗ y↑
9	id– num∗ id⟨expr′⟩	x – 2 ∗ y↑
5	id– num∗ id	x – 2 ∗ y↑

The next symbol determined each choice correctly.

Back to left-recursion elimination

Given a left-factored CFG, to eliminate left-recursion:

if $\exists\, A \rightarrow A\alpha$ then replace all of the A productions

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

with

$$A \rightarrow NA'$$
$$N \rightarrow \beta \mid \ldots \mid \gamma$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

where N and A' are new productions.

Repeat until there are no left-recursive productions.

Generality

Question:

> By *left factoring* and *eliminating left-recursion*, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

> Given a context-free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many *context-free languages* do not have such a grammar:

$$\{a^n 0 b^n \mid n \geq 1\} \bigcup \{a^n 1 b^{2n} \mid n \geq 1\}$$

Must look past an arbitrary number of a's to discover the 0 or the 1 and so determine the derivation.

Recursive descent parsing

Now, we can produce a simple recursive descent parser from the (right-associative) grammar.

```
goal:
   token ← next_token();
   if (expr() = ERROR | token ≠ EOF) then
      return ERROR;
expr:
   if (term() = ERROR) then
      return ERROR;
   else return expr_prime();
expr_prime:
   if (token = PLUS) then
      token ← next_token();
      return expr();
   else if (token = MINUS) then
      token ← next_token();
      return expr();
   else return OK;
```

Recursive descent parsing

```
term:
   if (factor() = ERROR) then
      return ERROR;
   else return term_prime();
term_prime:
   if (token = MULT) then
      token ← next_token();
      return term();
   else if (token = DIV) then
      token ← next_token();
      return term();
   else return OK;
factor:
   if (token = NUM) then
      token ← next_token();
      return OK;
   else if (token = ID) then
      token ← next_token();
      return OK;
   else return ERROR;
```

Building the tree

One of the key jobs of the parser is to build an intermediate representation of the source code.

To build an abstract syntax tree, we can simply insert code at the appropriate points:

- `factor()` can stack nodes `id`, `num`
- `term_prime()` can stack nodes $*$, $/$
- `term()` can pop 3, build and push subtree
- `expr_prime()` can stack nodes $+$, $-$
- `expr()` can pop 3, build and push subtree
- `goal()` can pop and return tree

Non-recursive predictive parsing

Observation:

> *Our recursive descent parser encodes state information in its run-time stack, or call stack.*

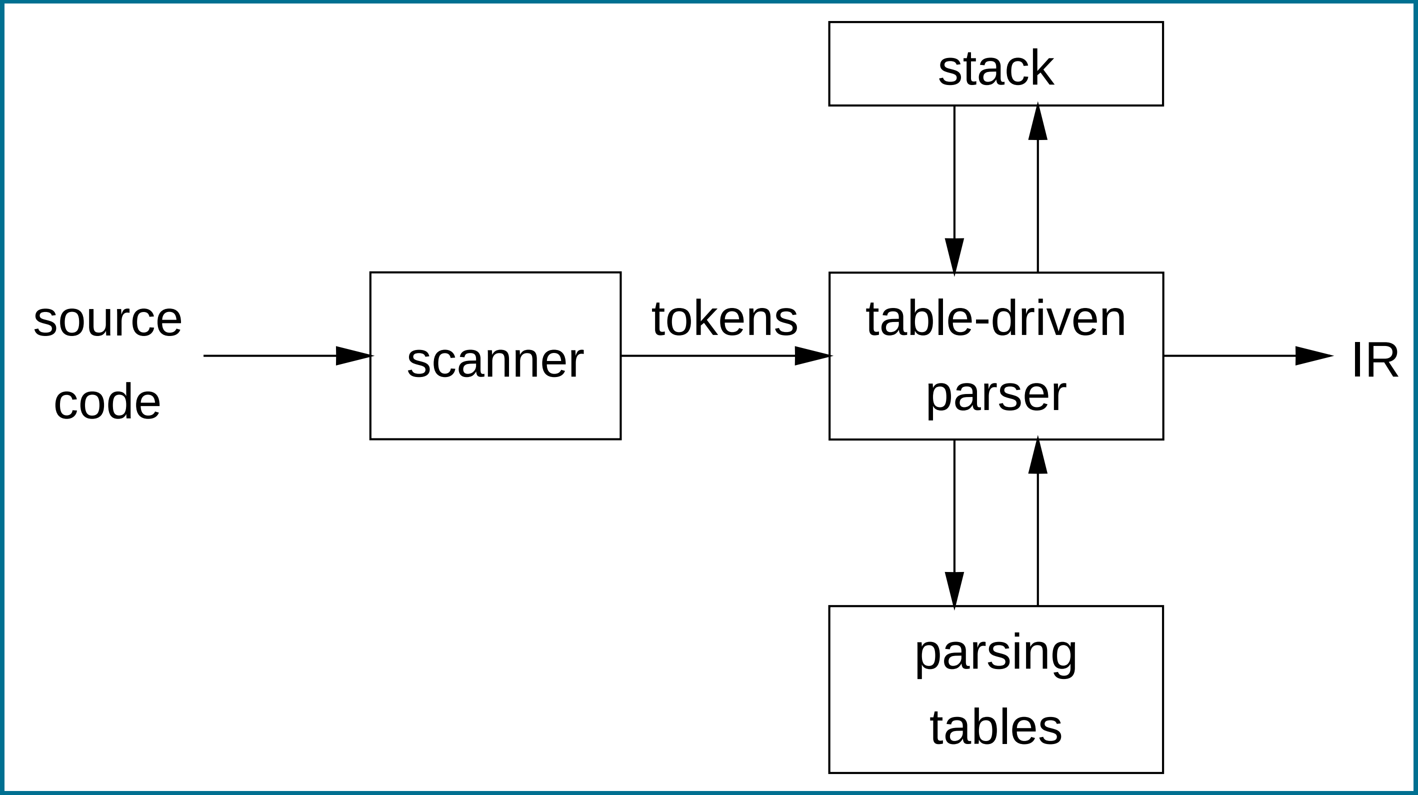
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

This suggests other implementation methods:

- explicit stack, hand-coded parser
- stack-based, table-driven parser

Non-recursive predictive parsing

Now, a predictive parser looks like:

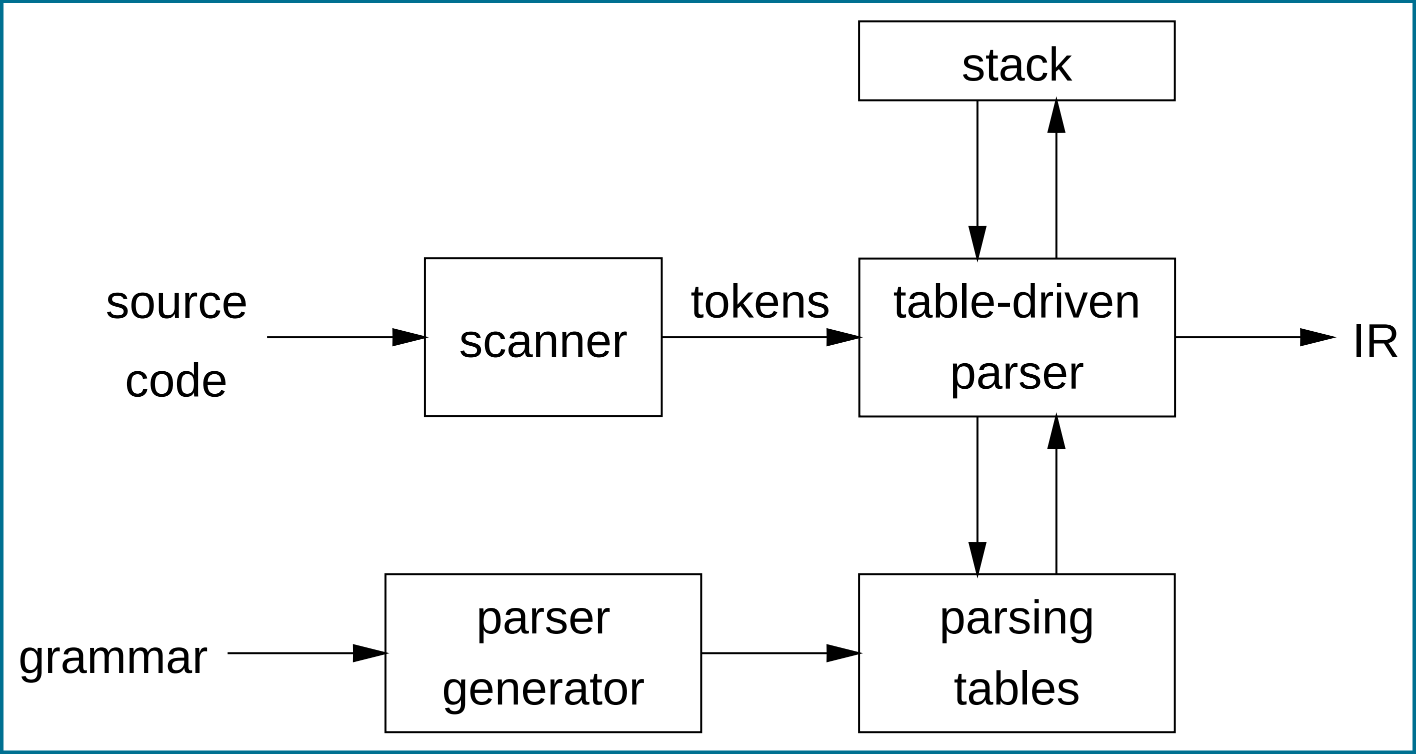


Rather than writing code, we build tables.

Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



This is true for both top-down (LL) and bottom-up (LR) parsers

Non-recursive predictive parsing

Input: a string w and a parsing table M for G

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
    X ← Stack[tos]
    if X is a terminal or EOF then
        if X = token then
            pop X
            token ← next_token()
        else error()
    else /* X is a non-terminal */
        if $M$[X,token] = $X \to Y_1 Y_2 \cdots Y_k$ then
            pop X
            push $Y_k, Y_{k-1}, \cdots, Y_1$
        else error()
until X = EOF
```

Non-recursive predictive parsing

What we need now is a parsing table M.

Our expression grammar:

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{term}\rangle\langle\text{expr}'\rangle$
3	$\langle\text{expr}'\rangle$	$::=$	$+\langle\text{expr}\rangle$
4		$\mid$	$-\langle\text{expr}\rangle$
5		$\mid$	ε
6	$\langle\text{term}\rangle$	$::=$	$\langle\text{factor}\rangle\langle\text{term}'\rangle$
7	$\langle\text{term}'\rangle$	$::=$	$*\langle\text{term}\rangle$
8		$\mid$	$/\langle\text{term}\rangle$
9		$\mid$	ε
10	$\langle\text{factor}\rangle$	$::=$	`num`
11		$\mid$	`id`

Its parse table:

	`id`	`num`	$+$	$-$	$*$	$/$	\$[†]
$\langle\text{goal}\rangle$	1	1	–	–	–	–	–
$\langle\text{expr}\rangle$	2	2	–	–	–	–	–
$\langle\text{expr}'\rangle$	–	–	3	4	–	–	5
$\langle\text{term}\rangle$	6	6	–	–	–	–	–
$\langle\text{term}'\rangle$	–	–	9	9	7	8	9
$\langle\text{factor}\rangle$	11	10	–	–	–	–	–

[†] we use \$ to represent `EOF`

FIRST

For a string of grammar symbols α, define $\textsc{first}(\alpha)$ as:

- the set of terminal symbols that begin strings derived from α:
  $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in \textsc{first}(\alpha)$

$\textsc{first}(\alpha)$ contains the set of tokens valid in the initial position in α

To build $\textsc{first}(X)$:

1. If $X \in V_t$ then $\textsc{first}(X)$ is $\{X\}$
2. If $X \rightarrow \varepsilon$ then add ε to $\textsc{first}(X)$.
3. If $X \rightarrow Y_1 Y_2 \cdots Y_k$:
   (a) Put $\textsc{first}(Y_1) - \{\varepsilon\}$ in $\textsc{first}(X)$
   (b) $\forall i : 1 < i \leq k$, if $\varepsilon \in \textsc{first}(Y_1) \cap \cdots \cap \textsc{first}(Y_{i-1})$
   (i.e., $Y_1 \cdots Y_{i-1} \Rightarrow^* \varepsilon$)
   then put $\textsc{first}(Y_i) - \{\varepsilon\}$ in $\textsc{first}(X)$
   (c) If $\varepsilon \in \textsc{first}(Y_1) \cap \cdots \cap \textsc{first}(Y_k)$ then put ε in $\textsc{first}(X)$

   Repeat until no more additions can be made.

FOLLOW

For a non-terminal A, define $\text{FOLLOW}(A)$ as

> the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set.

To build $\text{FOLLOW}(A)$:

1. Put \$ in $\text{FOLLOW}(\langle\text{goal}\rangle)$
2. If $A \rightarrow \alpha B \beta$:
   (a) Put $\text{FIRST}(\beta) - \{\varepsilon\}$ in $\text{FOLLOW}(B)$
   (b) If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in \text{FIRST}(\beta)$ (i.e., $\beta \Rightarrow^* \varepsilon$) then put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$

   Repeat until no more additions can be made

LL(1) grammars

Previous defi nition

A grammar G is LL(1) iff. for all non-terminals A, each distinct pair of productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition $\textsc{first}(\beta) \bigcap \textsc{first}(\gamma) = \phi$.

What if $A \Rightarrow^* \varepsilon$?

Revised defi nition

A grammar G is LL(1) iff. for each set of productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$:

1. $\textsc{first}(\alpha_1), \textsc{first}(\alpha_2), \ldots, \textsc{first}(\alpha_n)$ are all pairwise disjoint
2. If $\alpha_i \Rightarrow^* \varepsilon$ then $\textsc{first}(\alpha_j) \bigcap \textsc{follow}(A) = \phi, \forall 1 \leq j \leq n, i \neq j$.

If G is ε-free, condition 1 is sufficient.

LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example

$S \to aS \mid a$

is not LL(1) because $\textsc{first}(aS) = \textsc{first}(a) = \{a\}$

$S \to aS'$
$S' \to aS' \mid \varepsilon$

accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   (a) $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
   (b) If $\varepsilon \in \text{FIRST}(\alpha)$:
      i. $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
      ii. If $\$ \in \text{FOLLOW}(A)$ then add $A \rightarrow \alpha$ to $M[A, \$]$
2. Set each undefined entry of M to `error`

If $\exists M[A, a]$ with multiple entries then grammar is not LL(1).

Note: recall $a, b \in V_t$, so $a, b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$$\begin{array}{l|l} S \to E & T \to FT' \\ E \to TE' & T' \to *T \mid /T \mid \varepsilon \\ E' \to +E \mid -E \mid \varepsilon & F \to \texttt{id} \mid \texttt{num} \end{array}$$

	FIRST	FOLLOW
S	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E'	$\{\varepsilon, +, -\}$	$\{\$\}$
T	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, \$\}$
T'	$\{\varepsilon, *, /\}$	$\{+, -, \$\}$
F	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, *, /, \$\}$
id	$\{\texttt{id}\}$	$-$
num	$\{\texttt{num}\}$	$-$
$*$	$\{*\}$	$-$
$/$	$\{/\}$	$-$
$+$	$\{+\}$	$-$
$-$	$\{-\}$	$-$

	id	num	$+$	$-$	$*$	$/$	$\$$
S	$S \to E$	$S \to E$	$-$	$-$	$-$	$-$	$-$
E	$E \to TE'$	$E \to TE'$	$-$	$-$	$-$	$-$	$-$
E'	$-$	$-$	$E' \to +E$	$E' \to -E$	$-$	$-$	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	$-$	$-$	$-$	$-$	$-$
T'	$-$	$-$	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \texttt{id}$	$F \to \texttt{num}$	$-$	$-$	$-$	$-$	$-$

A grammar that is not LL(1)

$$\begin{array}{rcl} \langle\text{stmt}\rangle & ::= & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle \\ & \mid & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle\ \texttt{else}\ \langle\text{stmt}\rangle \\ & \mid & \ldots \end{array}$$

Left-factored:

$$\begin{array}{rcl} \langle\text{stmt}\rangle & ::= & \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle\ \langle\text{stmt}'\rangle \mid \ldots \\ \langle\text{stmt}'\rangle & ::= & \texttt{else}\ \langle\text{stmt}\rangle \mid \varepsilon \end{array}$$

Now, $\text{FIRST}(\langle\text{stmt}'\rangle) = \{\varepsilon, \texttt{else}\}$
Also, $\text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}, \$\}$
But, $\text{FIRST}(\langle\text{stmt}'\rangle) \bigcap \text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}\} \neq \phi$

On seeing `else`, conflict between choosing

$$\langle\text{stmt}'\rangle\ ::=\ \texttt{else}\ \langle\text{stmt}\rangle \quad \text{and} \quad \langle\text{stmt}'\rangle\ ::=\ \varepsilon$$

$\Rightarrow$ grammar is not LL(1)!

The fix:

Put priority on $\langle\text{stmt}'\rangle\ ::=\ \texttt{else}\ \langle\text{stmt}\rangle$ to associate `else` with closest previous `then`.

Error recovery

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize
- When an error occurs looking for A, scan until an element of $\textsc{synch}(A)$ is found

Building $\textsc{synch}$:

1. $a \in \textsc{follow}(A) \Rightarrow a \in \textsc{synch}(A)$
2. place keywords that start statements in $\textsc{synch}(A)$
3. add symbols in $\textsc{first}(A)$ to $\textsc{synch}(A)$

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

(i.e., $\textsc{synch}(a) = V_t - \{a\}$)

Chapter 4: LR Parsing

Some definitions

Recall

> For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a *sentential form*

- If $\alpha \in V_t^*$, then α is called a *sentence* in $L(G)$
- Otherwise it is just a sentential form (not a sentence in $L(G)$)

A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

Bottom-up parsing

Goal:

> *Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.*

The parser repeatedly matches a *right-sentential* form from the language against the tree's upper frontier.

At each match, it applies a *reduction* to build on the frontier:

- each reduction matches an upper frontier of the partially built tree to the RHS of some production
- each reduction adds a node on top of the frontier

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

1	S	$\rightarrow$	$\mathtt{a}AB\mathtt{e}$
2	A	$\rightarrow$	$A\mathtt{bc}$
3		$\mid$	$\mathtt{b}$
4	B	$\rightarrow$	$\mathtt{d}$

and the input string `abbcde`

Prod'n.	Sentential Form
3	$\mathtt{a}\boxed{\mathtt{b}}\mathtt{bcde}$
2	$\mathtt{a}\boxed{A\mathtt{bc}}\mathtt{de}$
4	$\mathtt{a}A\boxed{\mathtt{d}}\mathtt{e}$
1	$\boxed{\mathtt{a}AB\mathtt{e}}$
–	S

The trick appears to be scanning the input and finding valid sentential forms.

Handles

What are we trying to fi nd?

A substring α of the tree's upper frontier that

> matches some production $A \to \alpha$ where reducing α to A is one step in the reverse of a rightmost derivation

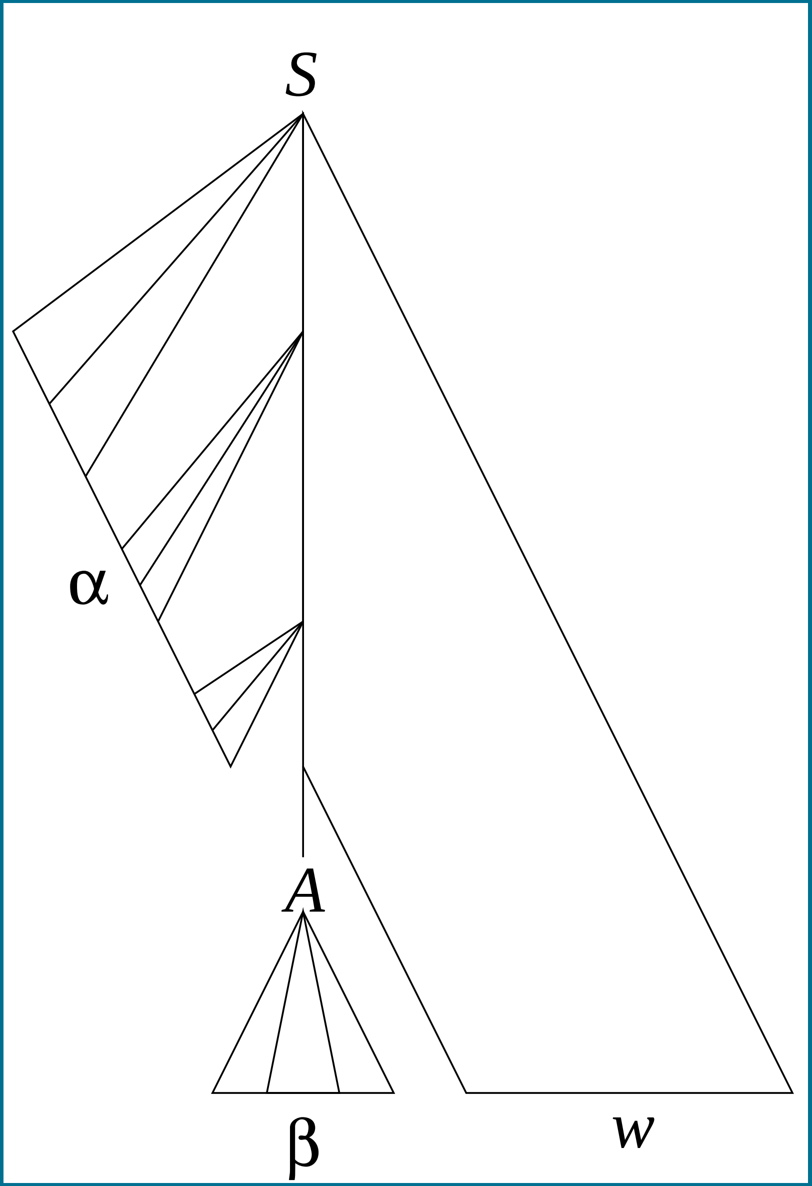
We call such a string a *handle*.

Formally:

> a *handle* of a right-sentential form γ is a production $A \to \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ
>
> i.e., if $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$ then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

Handles



The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

Handles

Theorem:

> If G is unambiguous then every right-sentential form has a unique handle.

Proof: (by defi nition)

1. G is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \to \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position k at which $A \to \beta$ is applied
4. $\Rightarrow$ a unique handle $A \to \beta$

Example

The left-recursive expression grammar (*original form*)

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\texttt{num}$
9		$\mid$	$\texttt{id}$

Prod'n.	Sentential Form
–	$\langle\text{goal}\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle\text{expr}\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning*.

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

we set i to n and apply the following simple algorithm

```
for i = $n$ downto 1
 1. find the handle $A_i \to \beta_i$ in $\gamma_i$
 2. replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$
```

This takes $2n$ steps, where n is the length of the derivation

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is $

   a) *fi nd the handle*
      if we don’t have a handle on top of the stack, *shift* an input symbol onto the stack

   b) *prune the handle*
      if we have a handle $A \to \beta$ on the stack, *reduce*

      i) pop $|\beta|$ symbols off the stack

      ii) push A onto the stack

Example: back to $\mathtt{x} - 2 * \mathtt{y}$

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\mathtt{num}$
9		$\mid$	$\mathtt{id}$

Stack	Input	Action
$\$$	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	shift
$\$\underline{\mathtt{id}}$	$- \mathtt{num} * \mathtt{id}$	reduce 9
$\$\underline{\langle\text{factor}\rangle}$	$- \mathtt{num} * \mathtt{id}$	reduce 7
$\$\underline{\langle\text{term}\rangle}$	$- \mathtt{num} * \mathtt{id}$	reduce 4
$\$\langle\text{expr}\rangle$	$- \mathtt{num} * \mathtt{id}$	shift
$\$\langle\text{expr}\rangle -$	$\mathtt{num} * \mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \underline{\mathtt{num}}$	$* \mathtt{id}$	reduce 8
$\$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle}$	$* \mathtt{id}$	reduce 7
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle$	$* \mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle *$	$\mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\mathtt{id}}$		reduce 9
$\$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$		reduce 5
$\$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$		reduce 3
$\$\underline{\langle\text{expr}\rangle}$		reduce 1
$\$\langle\text{goal}\rangle$		accept

1. *Shift until top of stack is the right end of a handle*

2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

Shift-reduce parsing

Shift-reduce parsers are simple to understand

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack

2. *reduce* — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal LHS

3. *accept* — terminate parsing and signal success

4. *error* — call an error recovery routine

The key problem: to recognize handles (not covered in this course).

LR(k) grammars

Informally, we say that a grammar G is LR(k) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form*, and
2. *determine the production by which to reduce*

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

LR(k) grammars

Formally, a grammar G is LR(k) iff.:

1. $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$, and
2. $S \Rightarrow^*_{\mathrm{rm}} \gamma B x \Rightarrow_{\mathrm{rm}} \alpha\beta y$, and
3. $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$

$\Rightarrow \alpha A y = \gamma B x$

i.e., Assume sentential forms $\alpha\beta w$ and $\alpha\beta y$, with common prefix $\alpha\beta$ and common k-symbol lookahead $\mathrm{FIRST}_k(y) = \mathrm{FIRST}_k(w)$, such that $\alpha\beta w$ reduces to $\alpha A w$ and $\alpha\beta y$ reduces to $\gamma B x$.

But, the common prefix means $\alpha\beta y$ also reduces to $\alpha A y$, for the same result.

Thus $\alpha A y = \gamma B x$.

Why study LR grammars?

LR(1) grammars are often used to construct parsers.

We call these parsers LR(1) parsers.

- everyone's favorite parser
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

  LL(k)**:** recognize use of a production $A \to \beta$ seeing first k symbols of β

  LR(k)**:** recognize occurrence of β (the handle) having seen all of what is derived from β plus k symbols of lookahead

Left versus right recursion

Right Recursion:

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:

- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

Parsing review

Recursive descent

A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k)

An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

LR(k)

An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

The dilemmas:

- LL dilemma: pick $A \rightarrow b$ or $A \rightarrow c$?
- LR dilemma: pick $A \rightarrow b$ or $B \rightarrow b$?

Chapter 5: JavaCC and JTB

The Java Compiler Compiler

- Can be thought of as “Lex and Yacc for Java.”
- It is based on LL(k) rather than LALR(1).
- Grammars are written in EBNF.
- The Java Compiler Compiler transforms an EBNF grammar into an LL(k) parser.
- The JavaCC grammar can have embedded action code written in Java, just like a Yacc grammar can have embedded action code written in C.
- The lookahead can be changed by writing `LOOKAHEAD(...)`.
- The whole input is given in just one file (not two).

The JavaCC input format

One file:

- header
- token specifications for lexical analysis
- grammar

The JavaCC input format

Example of a token specification:

```
TOKEN :
{
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
}
```

Example of a production:

```
void StatementListReturn() :
{}
{
  ( Statement() )* "return" Expression() ";"
}
```

Generating a parser with JavaCC

```
javacc fortran.jj   // generates a parser with a specified name
javac Main.java     // Main.java contains a call of the parser
java Main < prog.f  // parses the program prog.f
```

The Visitor Pattern

For **object-oriented programming**,

the Visitor pattern **enables**

the definition of a **new operation**

on an **object structure**

without changing the classes

of the objects.

Gamma, Helm, Johnson, Vlissides:
Design Patterns, 1995.

Sneak Preview

When using the **Visitor** pattern,

- the set of classes must be fixed in advance, and
- each class must have an accept method.

First Approach: Instanceof and Type Casts

The running Java example: summing an integer list.

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

First Approach: Instanceof and Type Casts

```
List l;      // The List-object
int sum = 0;
boolean proceed = true;
while (proceed) {
  if (l instanceof Nil)
     proceed = false;
  else if (l instanceof Cons) {
     sum = sum + ((Cons) l).head;
     l = ((Cons) l).tail;
     // Notice the two type casts!
  }
}
```

Advantage: The code is written without touching the classes `Nil` and `Cons`.

Drawback: The code constantly uses type casts and `instanceof` to determine what class of object it is considering.

Second Approach: Dedicated Methods

The first approach is **not** object-oriented!

To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects.

```
interface List {
  int sum();
}
```

We can now compute the sum of all components of a given `List`-object `l` by writing `l.sum()`.

Second Approach: Dedicated Methods

```
class Nil implements List {
  public int sum() {
    return 0;
  }
}

class Cons implements List {
  int head;
  List tail;
  public int sum() {
    return head + tail.sum();
  }
}
```

Advantage: The type casts and `instanceof` operations have disappeared, and the code can be written in a systematic way.

Disadvantage: For each new operation on `List`-objects, new dedicated methods have to be written, and all classes must be recompiled.

Third Approach: The Visitor Pattern

The Idea:

- Divide the code into an object structure and a Visitor (akin to Functional Programming!)
- Insert an `accept` method in each class. Each accept method takes a Visitor as argument.
- A Visitor contains a `visit` method for each class (overloading!) A method for a class C takes an argument of type C.

```
interface List {
  void accept(Visitor v);
}

interface Visitor {
  void visit(Nil x);
  void visit(Cons x);
}
```

Third Approach: The Visitor Pattern

- The purpose of the `accept` methods is to invoke the `visit` method in the Visitor which can handle the current object.

```
class Nil implements List {
  public void accept(Visitor v) {
    v.visit(this);
  }
}

class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

Third Approach: The Visitor Pattern

- The control flow goes back and forth between the `visit` methods in the Visitor and the `accept` methods in the object structure.

```
class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    sum = sum + x.head;
    x.tail.accept(this);
  }
}
.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Notice: The `visit` methods describe both
1) actions, and 2) access of subobjects.

Comparison

The Visitor pattern combines the advantages of the two other approaches.

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

The advantage of Visitors: New methods without recompilation!
Requirement for using Visitors: All classes must have an accept method.

Tools that use the Visitor pattern:

- JJTree (from Sun Microsystems) and the Java Tree Builder (from Purdue University), both frontends for The Java Compiler Compiler from Sun Microsystems.

Visitors: Summary

- **Visitor makes adding new operations easy.** Simply write a new visitor.
- **A visitor gathers related operations.** It also separates unrelated ones.
- **Adding new classes to the object structure is hard.** Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most like to change the classes of objects that make up the structure.
- **Visitors can accumulate state.**
- **Visitor can break encapsulation.** Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

The Java Tree Builder

The Java Tree Builder (JTB) has been developed here at Purdue in my group.

JTB is a frontend for The Java Compiler Compiler.

JTB supports the building of syntax trees which can be traversed using visitors.

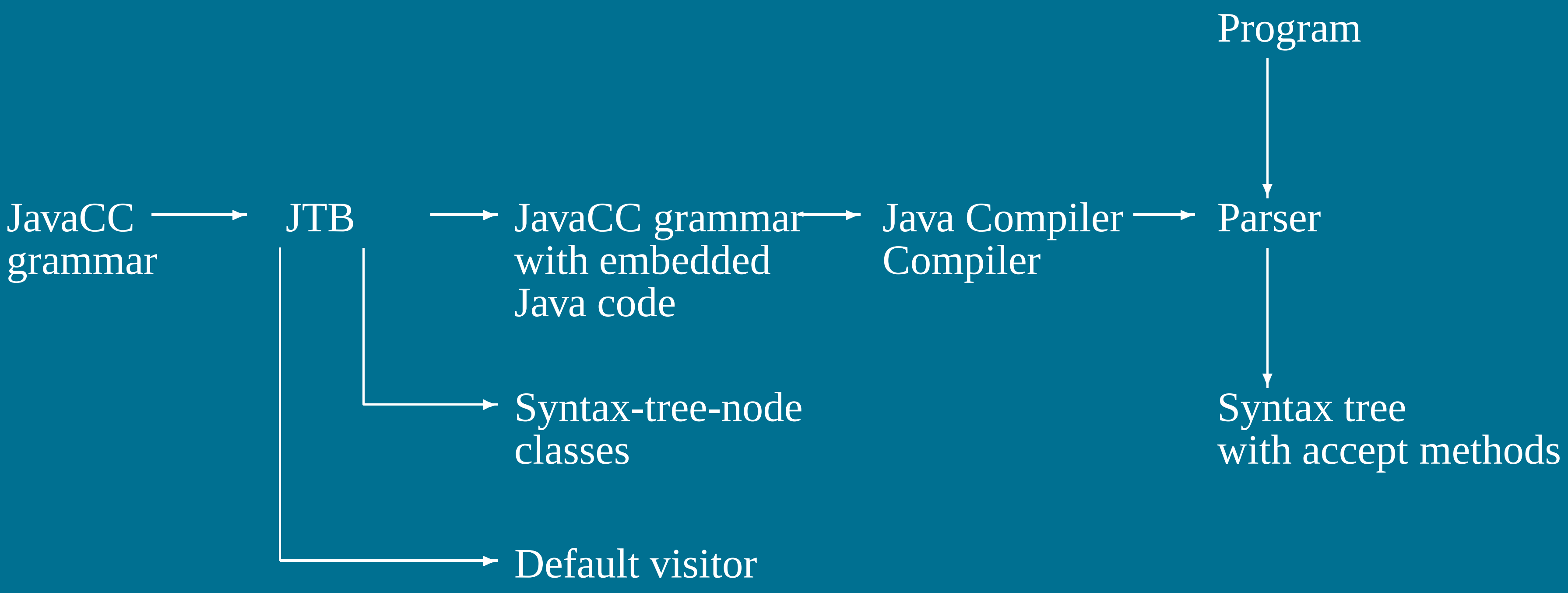
JTB transforms a bare JavaCC grammar into three components:

- a JavaCC grammar with embedded Java code for building a syntax tree;
- one class for every form of syntax tree node; and
- a default visitor which can do a depth-first traversal of a syntax tree.

The Java Tree Builder

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.

The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Using JTB

```
jtb fortran.jj        // generates jtb.out.jj
javacc jtb.out.jj     // generates a parser with a specified name
javac Main.java       // Main.java contains a call of the parser
                         and calls to visitors
java Main < prog.f    // builds a syntax tree for prog.f, and
                         executes the visitors
```

Example (simplified)

For example, consider the Java 1.1 production

```
void Assignment() : {}
   { PrimaryExpression() AssignmentOperator()
     Expression() }
```

JTB produces:

```
Assignment Assignment () :
{ PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {} }
{ n0=PrimaryExpression()
  n1=AssignmentOperator()
  n2=Expression()
  { return new Assignment(n0,n1,n2); }
}
```

Notice that the production returns a syntax tree represented as an `Assignment` object.

Example (simplified)

JTB produces a syntax-tree-node class for `Assignment`:

```
public class Assignment implements Node {
  PrimaryExpression f0; AssignmentOperator f1;
  Expression f2;

  public Assignment(PrimaryExpression n0,
                    AssignmentOperator n1,
                    Expression n2)
  { f0 = n0; f1 = n1; f2 = n2; }

  public void accept(visitor.Visitor v) {
      v.visit(this);
  }
}
```

Notice the `accept` method; it invokes the method `visit` for `Assignment` in the default visitor.

Example (simplified)

The default visitor looks like this:

```
public class DepthFirstVisitor implements Visitor {
  ...
   //
   // f0 -> PrimaryExpression()
   // f1 -> AssignmentOperator()
   // f2 -> Expression()
   //
   public void visit(Assignment n) {
      n.f0.accept(this);
      n.f1.accept(this);
      n.f2.accept(this);
   }
}
```

Notice the body of the method which visits each of the three subtrees of the `Assignment` node.

Example (simplified)

Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```
public class VprintAssignRHS extends DepthFirstVisitor {
   void visit(Assignment n) {
      VPrettyPrinter v = new VPrettyPrinter();
      n.f2.accept(v); v.out.println();
      n.f2.accept(this);
   }
}
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visit` in `VprintAssignRHS` is executed.

Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.

JTB is bootstrapped.

Chapter 6: Semantic Analysis

Semantic Analysis

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or subtrees of a syntax tree

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is `x` scalar, an array, or a function?
2. Is `x` declared before it is used?
3. Are any names declared but not used?
4. Which declaration of `x` does this reference?
5. Is an expression *type-consistent*?
6. Does the dimension of a reference match the declaration?
7. Where can `x` be stored? (heap, stack, ...)
8. Does `*p` reference the result of a malloc()?
9. Is `x` defined before it is used?
10. Is an array reference *in bounds*?
11. Does function `foo` produce a constant value?
12. Can `p` be implemented as a *memo-function*?

These cannot be answered with a context-free grammar

Context-sensitive analysis

Why is context-sensitive analysis hard?

- answers depend on values, not syntax
- questions and answers involve non-local information
- answers may involve computation

Several alternatives:

abstract syntax tree (*attribute grammars*)	specify non-local computations automatic evaluators
symbol tables	central store for facts express checking code
language design	simplify language avoid problems

Symbol tables

For *compile-time* efficiency, compilers often use a *symbol table*:

- associates lexical *names* (symbols) with their *attributes*

What items should be entered?

- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries (*we'll get there*)

Separate table for structure layouts (types) (*fi eld offsets and lengths*)

A symbol table is a compile-time structure

Symbol table information

What kind of information might the compiler need?

- textual name
- data type
- dimension information (*for aggregates*)
- declaring procedure
- lexical level of declaration
- storage class (*base address*)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

Nested scopes: block-structured symbol tables

What information is needed?

- when we ask about a name, we want the *most recent* declaration
- the declaration may be from the current scope or some enclosing scope
- innermost scope overrides declarations from outer scopes

Key point: new declarations (usually) occur only in current scope

What operations do we need?

- `void put (Symbol key, Object value)` – binds key to value
- `Object get(Symbol key)` – returns value bound to key
- `void beginScope()` – remembers current state of table
- `void endScope()` – restores table to state at most recent scope that has not been ended

May need to preserve list of locals for the debugger

Attribute information

Attributes are internal representation of declarations

Symbol table associates names with attributes

Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

Type expressions

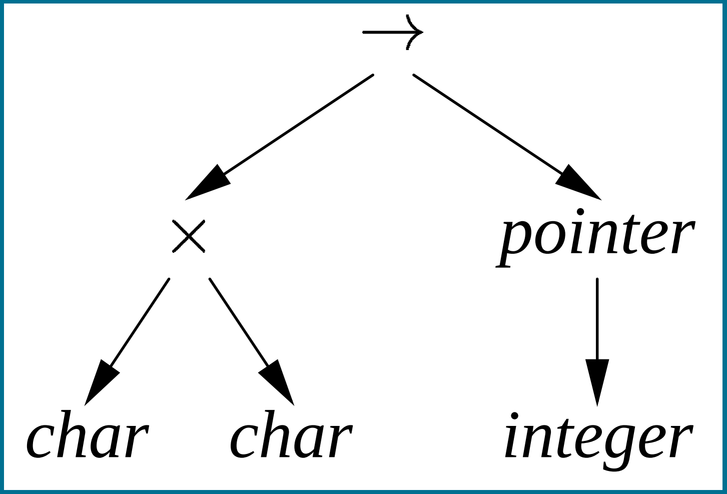
Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   (a) $array(I,T)$ denotes array of elements type T, index type I
   e.g., $array(1\dots10, integer)$
   (b) $T_1 \times T_2$ denotes Cartesian product of type expressions T_1 and T_2
   (c) records: fields have names
   e.g., $record((\mathtt{a} \times integer), (\mathtt{b} \times real))$
   (d) $pointer(T)$ denotes the type "pointer to object of type T"
   (e) $D \rightarrow R$ denotes type of function mapping domain D to range R
   e.g., $integer \times integer \rightarrow integer$

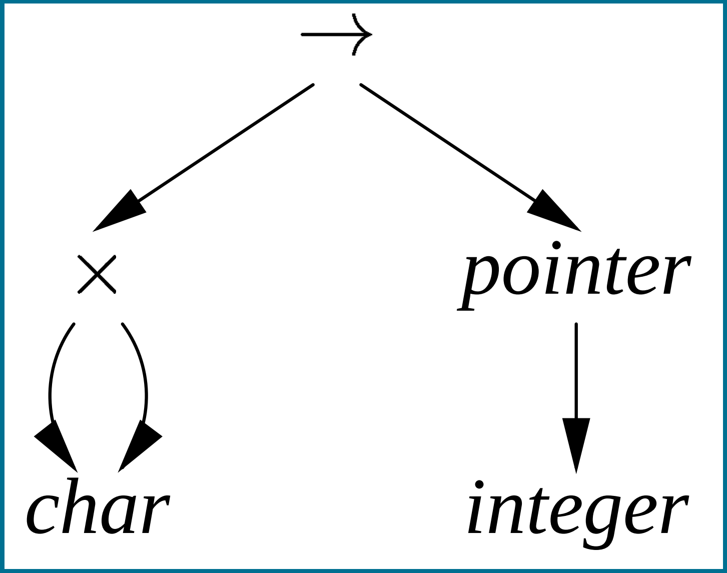
Type descriptors

Type descriptors are compile-time structures representing type expressions

e.g., $char \times char \rightarrow pointer(integer)$



or



Type compatibility

Type checking needs to determine type equivalence

Two approaches:

Name equivalence: each type name is a distinct type

Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names)

- $s \equiv t$ iff. s and t are the same basic types
- $array(s_1, s_2) \equiv array(t_1, t_2)$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $pointer(s) \equiv pointer(t)$ iff. $s \equiv t$
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Type compatibility: example

Consider:

```
type  link  =  ↑cell;
var   next  :  link;
      last  :  link;
      p     :  ↑cell;
      q, r  :  ↑cell;
```

Under name equivalence:

- `next` and `last` have the same type
- `p`, `q` and `r` have the same type
- `p` and `next` have different type

Under structural equivalence all variables have the same type

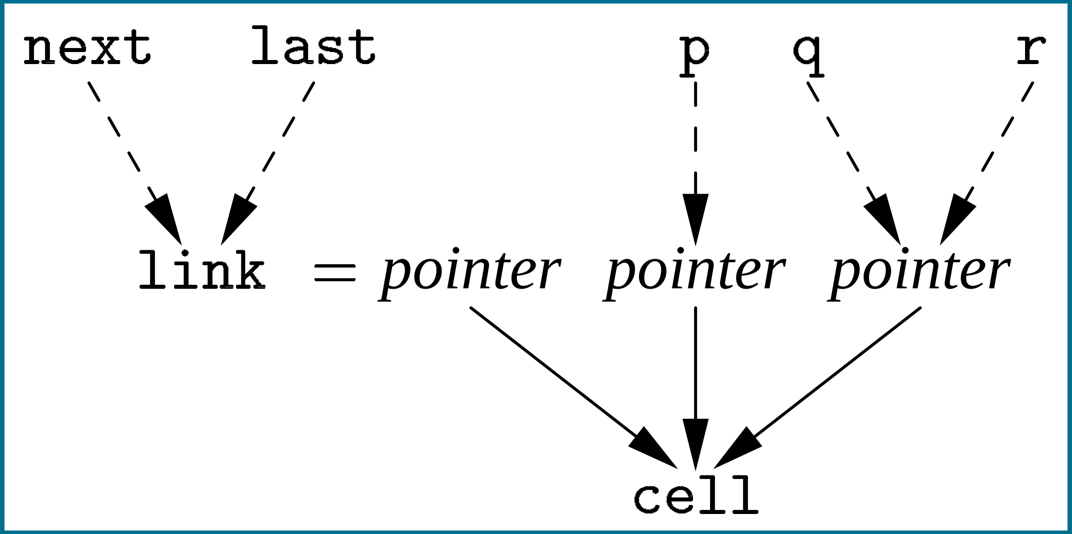
Ada/Pascal/Modula-2 are somewhat confusing: they treat distinct type definitions as distinct types, so

`p` has different type from `q` and `r`

Type compatibility: Pascal-style name equivalence

Build compile-time structure called a *type graph*:

- each constructor or basic type creates a node
- each name creates a leaf (associated with the type's descriptor)



Type expressions are equivalent if they are represented by the same node in the graph

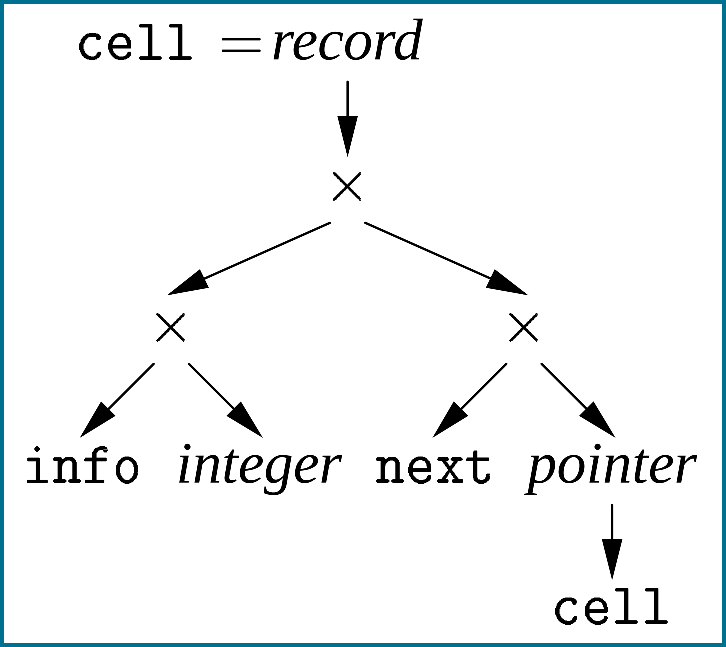
Type compatibility: recursive types

Consider:

```
 type  link  =  ↑cell;
       cell  =  record
                info :  integer;
                next :  link;
                end;
```

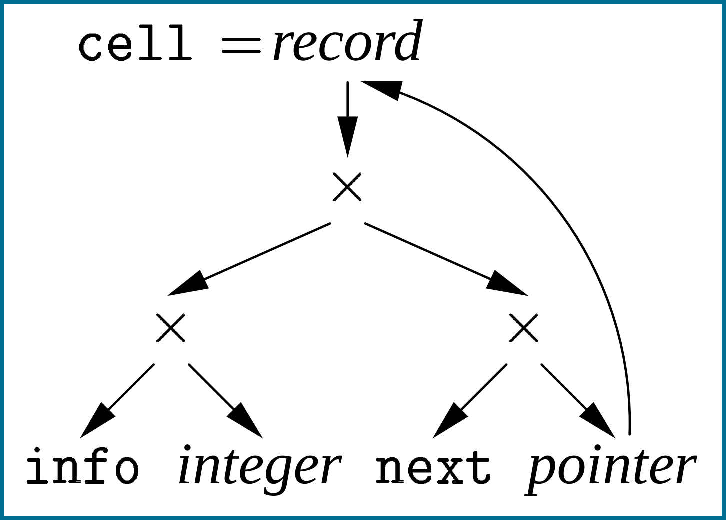
We may want to eliminate the names from the type graph

Eliminating name `link` from type graph for record:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:

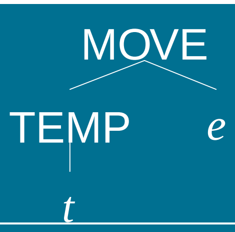
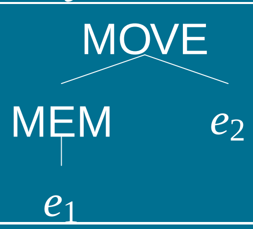
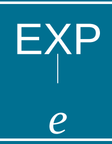
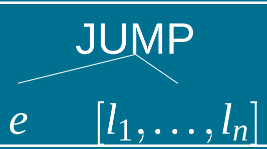
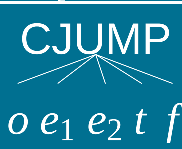
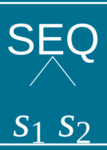
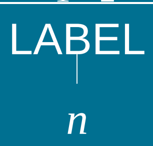


Chapter 7: Translation and Simplification

IR trees: Expressions

CONST i	Integer constant i	
NAME n	Symbolic constant n	[a code label]
TEMP t	Temporary t	[one of any number of “registers”]
BINOP $o\ e_1\ e_2$	Application of binary operator o: PLUS, MINUS, MUL, DIV AND, OR, XOR LSHIFT, RSHIFT ARSHIFT to integer operands e_1 (evaluated fi rst) and e_2 (evaluated second)	 [bitwise logical] [logical shifts] [arithmetic right-shift]
MEM e	Contents of a word of memory starting at address e	
CALL $f\ [e_1,\ldots,e_n]$	Procedure call; expression f is evaluated before arguments $e_1,\ldots,e_n$	
ESEQ $s\ e$	Expression sequence; evaluate s for side-effects, then e for result	

IR trees: Statements

	Evaluate e into temporary t
	Evaluate e_1 yielding address a, e_2 into word at a
	Evaluate e and discard result
	Transfer control to address e; $l_1, \ldots, l_n$ are all possible values for e
	Evaluate e_1 then e_2, yielding a and b, respectively; compare a with b using relational operator o: EQ, NE [signed and unsigned integers] LT, GT, LE, GE [signed] ULT, ULE, UGT, UGE [unsigned] jump to t if true, f if false
	Statement s_1 followed by s_2
	Defi ne constant value of name n as current code address; $\mathrm{NAME}(n)$ can be used as target of jumps, calls, etc.

Kinds of expressions

Expression kinds indicate “how expression might be used”

Ex(exp) expressions that compute a value

Nx(stm) statements: expressions that compute no value

Cx conditionals (jump to true and false destinations)

  RelCx(op, left, right)

IfThenElseExp expression/statement depending on use

Conversion operators allow use of one form in context of another:

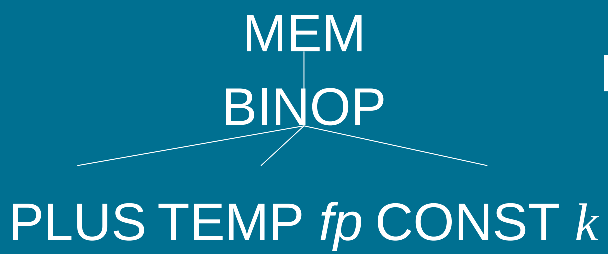
unEx convert to tree expression that computes value of inner tree

unNx convert to tree statement that computes inner tree but returns no value

unCx(t, f) convert to statement that evaluates inner tree and branches to true destination if non-zero, false destination otherwise

Translating

Simple variables: fetch with a MEM:



Ex(MEM($+$(TEMP *fp*, CONST k)))

where *fp* is home frame of variable, found by following static links; k is offset of variable in that level

Array variables: Suppose arrays are pointers to array base. So fetch with a MEM like any other variable:

Ex(MEM($+$(TEMP *fp*, CONST k)))

Thus, for $e[i]$:

Ex(MEM($+$(e.unEx, $\times$(i.unEx, CONST w))))

i is index expression and w is word size

Note: must first check array index $i < \mathrm{size}(\mathrm{e})$; runtime will put size in word preceding array base

Translating

Record variables: Suppose records are pointers to record base, so fetch like other variables. For e.`f`:

Ex(MEM($+$(e.unEx, CONST o)))

where o is the byte offset of the fi eld `f` in the record
Note: must check record pointer is non-nil (i.e., non-zero)

String literals: Statically allocated, so just use the string's label

Ex(NAME(*label*))

where the literal will be emitted as:

```
              .word 11
label:        .ascii "hello world"
```

Record creation: `t`$\{f_1 = e_1, f_2 = e_2, \ldots f_n = e_n\}$ in the (preferably GC'd) heap, fi rst allocate the space then initialize it:

```
Ex( ESEQ(SEQ(MOVE(TEMP r, externalCall(”allocRecord”, [CONST $n$])),
             SEQ(MOVE(MEM(TEMP r), $e_1$.unEx)),
                 SEQ(. . . ,
                     MOVE(MEM(+(TEMP r, CONST $(n-1)w$)),
                          $e_n$.unEx))),
        TEMP r))
```

where w is the word size

Array creation: `t`$[e_1]$ `of` e_2: Ex(externalCall(”initArray”, [e_1.unEx, e_2.unEx]))

Control structures

Basic blocks:

- a sequence of straight-line code
- if one instruction executes then they all execute
- a maximal sequence of instructions without branches
- a label starts a new basic block

Overview of control structure translation:

- control flow links up the basic blocks
- ideas are simple
- implementation requires bookkeeping
- some care is needed for good code

while loops

while c **do** s:

1. evaluate c
2. if false jump to next statement after loop
3. if true fall into loop body
4. branch to top of loop

e.g.,

```
    test:
            if not(c) jump done
            s
            jump test
    done:
    Nx( SEQ(SEQ(SEQ(LABEL test, c.unCx(body, done)),
                    SEQ(SEQ(LABEL body, s.unNx), JUMP(NAME test))),
              LABEL done))
```

repeat e_1 **until** e_2 $\Rightarrow$ evaluate/compare/branch at bottom of loop

for loops

for `i` := e_1 **to** e_2 **do** s

1. evaluate lower bound into index variable
2. evaluate upper bound into limit variable
3. if index $>$ limit jump to next statement after loop
4. fall through to loop body
5. increment index
6. if index $\leq$ limit jump to top of loop body

```
        $t_1 \leftarrow e_1$
        $t_2 \leftarrow e_2$
        if $t_1 > t_2$ jump *done*
*body*: $s$
        $t_1 \leftarrow t_1 + 1$
        if $t_1 \leq t_2$ jump *body*
*done*:
```

For **break** statements:

- when translating a loop push the *done* label on some stack
- **break** simply jumps to label on top of stack
- when done translating loop and its body, pop the label

Function calls

$f(e_1,\ldots,e_n)$:

Ex(CALL(NAME *label* $_f$, [*sl*,e_1,. . . e_n]))

where *sl* is the static link for the callee f, found by following n static links from the caller, n being the difference between the levels of the caller and the callee

Comparisons

Translate a *op* b as:

RelCx(*op*, a.unEx, b.unEx)

When used as a conditional unCx(t, f) yields:

CJUMP(*op*, a.unEx, b.unEx, t, f)

where t and f are labels.

When used as a value unEx yields:

```
ESEQ(SEQ(MOVE(TEMP r, CONST 1),
             SEQ(unCx(t, f),
                  SEQ(LABEL f,
                       SEQ(MOVE(TEMP r, CONST 0), LABEL t)))),
       TEMP r)
```

Conditionals

The short-circuiting Boolean operators have already been transformed into if-expressions in the abstract syntax:

e.g., $x < 5$ & $a > b$ turns into **if** $x < 5$ **then** $a > b$ **else** 0

Translate **if** e_1 **then** e_2 **else** e_3 into: IfThenElseExp(e_1, e_2, e_3)

When used as a value unEx yields:

```
ESEQ(SEQ(SEQ(e1.unCx(t, f),
                 SEQ(SEQ(LABEL t,
                         SEQ(MOVE(TEMP r, e2.unEx),
                             JUMP join)),
                     SEQ(LABEL f,
                         SEQ(MOVE(TEMP r, e3.unEx),
                             JUMP join)))),
             LABEL join),
     TEMP r)
```

As a conditional unCx(t, f) yields:

```
SEQ(e1.unCx(tt,ff), SEQ(SEQ(LABEL tt, e2.unCx(t, f)),
                        SEQ(LABEL ff, e3.unCx(t, f))))
```

Conditionals: Example

Applying unCx(t, f) to **if** $x < 5$ **then** $a > b$ **else** 0:

SEQ(CJUMP(LT, x.unEx, CONST 5, tt, ff),
        SEQ(SEQ(LABEL tt, CJUMP(GT, a.unEx, b.unEx, t, f)),
                SEQ(LABEL ff, JUMP f)))

or more optimally:

SEQ(CJUMP(LT, x.unEx, CONST 5, tt, f),
        SEQ(LABEL tt, CJUMP(GT, a.unEx, b.uneX, t, f)))

One-dimensional fixed arrays

```
var a : ARRAY [2..5] of integer;
...
a[e]
```

translates to:

MEM(+(TEMP fp, +(CONST $k-2w$, $\times$(CONST w, e.unEx))))

where k is offset of static array from fp, w is word size

In Pascal, multidimensional arrays are treated as arrays of arrays, so `A[i,j]` is equivalent to A[i][j], so can translate as above.

Multidimensional arrays

Array allocation:

constant bounds

- allocate in static area, stack, or heap
- no run-time descriptor is needed

dynamic arrays: bounds fixed at run-time

- allocate in stack or heap
- descriptor is needed

dynamic arrays: bounds can change at run-time

- allocate in heap
- descriptor is needed

Multidimensional arrays

Array layout:

- *Contiguous*:
  1. *Row major*

     Rightmost subscript varies most quickly:

     ```
     A[1,1], A[1,2], ...
     A[2,1], A[2,2], ...
     ```

     Used in PL/1, Algol, Pascal, C, Ada, Modula-3
  2. *Column major*

     Leftmost subscript varies most quickly:

     ```
     A[1,1], A[2,1], ...
     A[1,2], A[2,2], ...
     ```

     Used in FORTRAN
- *By vectors*

  Contiguous vector of *pointers* to (non-contiguous) subarrays

Multi-dimensional arrays: row-major layout

```
array [1..N,1..M] of T
≡ array [1..N] of array [1..M] of T
```

no. of elt's in dimension j:

$$D_j = U_j - L_j + 1$$

position of `A[`i_1`, ..., `i_n`]`:

$$\begin{array}{l} (i_n - L_n) \\ +(i_{n-1} - L_{n-1})D_n \\ +(i_{n-2} - L_{n-2})D_n D_{n-1} \\ +\cdots \\ +(i_1 - L_1)D_n \cdots D_2 \end{array}$$

which can be rewritten as

$$\overbrace{i_1 D_2 \cdots D_n + i_2 D_3 \cdots D_n + \cdots + i_{n-1} D_n + i_n}^{\text{variable part}} \\ - \underbrace{(L_1 D_2 \cdots D_n + L_2 D_3 \cdots D_n + \cdots + L_{n-1} D_n + L_n)}_{\text{constant part}}$$

address of `A[`i_1`, ..., `i_n`]`:

address(`A`) + ((variable part $-$ constant part) $\times$ element size)

case statements

case E **of** V_1: $S_1 \ldots V_n$: S_n **end**

1. evaluate the expression
2. find value in case list equal to value of expression
3. execute statement associated with value found
4. jump to next statement after case

Key issue: finding the right case

- sequence of conditional jumps (small case set)
  $\mathbf{O}(|\text{cases}|)$
- binary search of an ordered jump table (sparse case set)
  $\mathbf{O}(\log_2 |\text{cases}|)$
- hash table (dense case set)
  $\mathbf{O}(1)$

case statements

case E **of** V_1: $S_1 \ldots V_n$: S_n **end**

One translation approach:

```
        $t$ := *expr*
        jump test
$L_1$:  code for $S_1$
        jump next
$L_2$:  code for $S_2$
        jump next
        $\ldots$
$L_n$:  code for $S_n$
        jump next
test:   if $t = V_1$ jump $L_1$
        if $t = V_2$ jump $L_2$
        $\ldots$
        if $t = V_n$ jump $L_n$
        code to raise run-time exception
next:
```

Simplification

- Goal 1: No SEQ or ESEQ.
- Goal 2: CALL can only be subtree of EXP(. . .) or MOVE(TEMP t,. . .).

Transformations:

- lift ESEQs up tree until they can become SEQs
- turn SEQs into linear list

$$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) = \text{ESEQ}(\text{SEQ}(s_1,s_2), e)$$

$$\text{BINOP}(\mathit{op}, \text{ESEQ}(s, e_1), e_2) = \text{ESEQ}(s, \text{BINOP}(\mathit{op}, e_1, e_2))$$

$$\text{MEM}(\text{ESEQ}(s, e_1)) = \text{ESEQ}(s, \text{MEM}(e_1))$$

$$\text{JUMP}(\text{ESEQ}(s, e_1)) = \text{SEQ}(s, \text{JUMP}(e_1))$$

$$\text{CJUMP}(\mathit{op}, \text{ESEQ}(s, e_1), e_2, l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(\mathit{op}, e_1, e_2, l_1, l_2))$$

$$\text{BINOP}(\mathit{op}, e_1, \text{ESEQ}(s, e_2)) = \text{ESEQ}(\text{MOVE}(\text{TEMP t}, e_1), \text{ESEQ}(s, \text{BINOP}(\mathit{op}, \text{TEMP t}, e_2)))$$

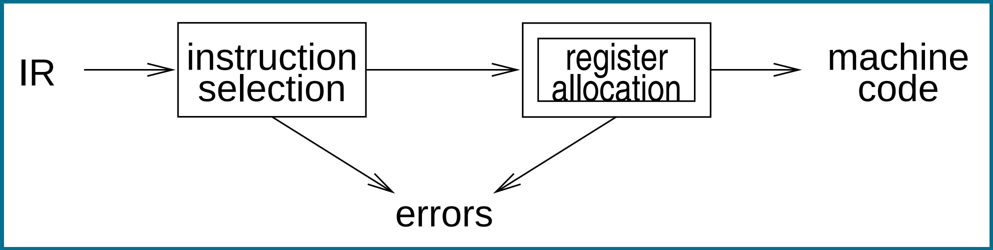
$$\text{CJUMP}(\mathit{op}, e_1, \text{ESEQ}(s, e_2), l_1, l_2) = \text{SEQ}(\text{MOVE}(\text{TEMP t}, e_1), \text{SEQ}(s, \text{CJUMP}(\mathit{op}, \text{TEMP t}, e_2, l_1, l_2)))$$

$$\text{MOVE}(\text{ESEQ}(s, e1), e_2) = \text{SEQ}(s, \text{MOVE}(e_1, e_2))$$

$$\text{CALL}(f, a) = \text{ESEQ}(\text{MOVE}(\text{TEMP t}, \text{CALL}(f, a)), \text{TEMP(t)})$$

Chapter 8: Liveness Analysis and Register Allocation

Register allocation



Register allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers

Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint *live* ranges can map to same register
- if not enough registers then *spill* some temporaries
  (i.e., keep them in memory)

The compiler must perform *liveness analysis* for each temporary:

It is *live* if it holds a value that may be needed in future

Control flow analysis

Before performing liveness analysis, need to understand the control flow by building a *control flow graph* (CFG):

- nodes may be individual program statements or basic blocks
- edges represent potential flow of control

Out-edges from node n lead to *successor* nodes, *succ*$[n]$
In-edges to node n come from *predecessor* nodes, *pred*$[n]$

Example:

```
      $a \leftarrow 0$
$L_1$ :  $b \leftarrow a + 1$
      $c \leftarrow c + b$
      $a \leftarrow b \times 2$
      if $a < N$ goto $L_1$
      return $c$
```

Liveness analysis

Gathering liveness information is a form of *data flow analysis* operating over the CFG:

- liveness of variables “flows” around the edges of the graph
- assignments *defi ne* a variable, v:
  - $\mathit{def}(v) = \text{set of graph nodes that define } v$
  - $\mathit{def}[n] = \text{set of variables defined by } n$
- occurrences of v in expressions *use* it:
  - $\mathit{use}(v) = \text{set of nodes that use } v$
  - $\mathit{use}[n] = \text{set of variables used in } n$

Liveness: v is *live* on edge e if there is a directed path from e to a *use* of v that does not pass through any $\mathit{def}(v)$

v is *live-in* at node n if live on any of n’s in-edges

v is *live-out* at n if live on any of n’s out-edges

$v \in \mathit{use}[n] \Rightarrow v \text{ live-in at } n$

$v \text{ live-in at } n \Rightarrow v \text{ live-out at all } m \in \mathit{pred}[n]$

$v \text{ live-out at } n, v \notin \mathit{def}[n] \Rightarrow v \text{ live-in at } n$

Liveness analysis

Define:

$\mathit{in}[n]$: variables live-in at n

$\mathit{in}[n]$: variables live-out at n

Then:

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}(n)} \mathit{in}[s]$$

$$\mathit{succ}[n] = \phi \Rightarrow \mathit{out}[n] = \phi$$

Note:

$$\mathit{in}[n] \supseteq \mathit{use}[n]$$

$$\mathit{in}[n] \supseteq \mathit{out}[n] - \mathit{def}[n]$$

$\mathit{use}[n]$ and $\mathit{def}[n]$ are constant (independent of control flow)

Now, $v \in \mathit{in}[n]$ iff. $v \in \mathit{use}[n]$ or $v \in \mathit{out}[n] - \mathit{def}[n]$

Thus, $\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$

Iterative solution for liveness

```
**foreach** $n$ $\{ in[n] \leftarrow \phi;\ out[n] \leftarrow \phi \}$
**repeat**
    **foreach** $n$
        $in'[n] \leftarrow in[n];$
        $out'[n] \leftarrow out[n];$
        $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
        $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
**until** $in'[n] = in[n] \wedge out'[n] = out[n], \forall n$
```

Notes:

- should order computation of inner loop to follow the “flow”
- liveness flows *backward* along control-flow arcs, from *out* to *in*
- nodes can just as easily be basic blocks to reduce CFG size
- could do one variable at a time, from *uses* back to *defs*, noting liveness along the way

Iterative solution for liveness

Complexity: for input program of size N

- $\leq N$ nodes in CFG
  $\Rightarrow \leq N$ variables
  $\Rightarrow N$ elements per *in/out*
  $\Rightarrow \mathrm{O}(N)$ time per set-union
- **for** loop performs constant number of set operations per node
  $\Rightarrow \mathrm{O}(N^2)$ time for **for** loop
- each iteration of **repeat** loop can only add to each set
  sets can contain at most every variable
  $\Rightarrow$ sizes of all in and out sets sum to $2N^2$,
  bounding the number of iterations of the **repeat** loop

$\Rightarrow$ worst-case complexity of $\mathrm{O}(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  $\Rightarrow \mathrm{O}(N)$ or $\mathrm{O}(N^2)$ in practice

Iterative solution for liveness

Least fi xed points

There is often more than one solution for a given dataflow problem (see example).

Any solution to dataflow equations is a *conservative approximation*:

- v has some later use downstream from n
  $\Rightarrow v \in \mathit{out}(n)$
- but not the converse

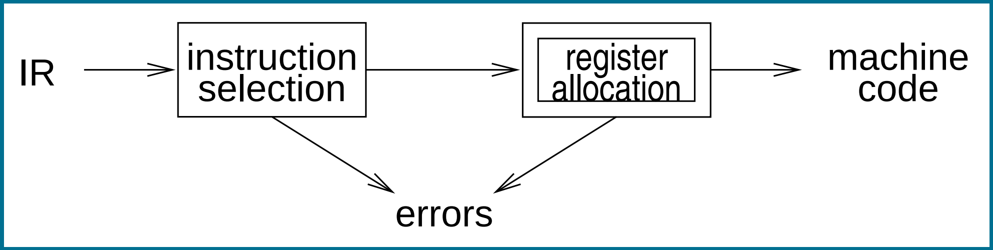
Conservatively assuming a variable is live does not break the program; just means more registers may be needed.

Assuming a variable is dead when it is really live *will* break things.

May be many possible solutions but want the “smallest”: the least fixpoint.

The iterative liveness computation computes this least fixpoint.

Register allocation



Register allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers

Register allocation by simplification

1. *Build* interference graph G: for each program point
   (a) compute set of temporaries simultaneously live
   (b) add edge to graph for each pair in set
2. *Simplify*: Color graph using a simple heuristic
   (a) suppose G has node m with degree $< K$
   (b) if $G' = G - \{m\}$ can be colored then so can G, since nodes adjacent to m have at most $K - 1$ colors
   (c) each such simplification will reduce degree of remaining nodes leading to more opportunity for simplification
   (d) leads to recursive coloring algorithm
3. *Spill*: suppose $\nexists m$ of degree $< K$
   (a) target some node (temporary) for spilling (optimistically, spilling node will allow coloring of remaining nodes)
   (b) remove and continue simplifying

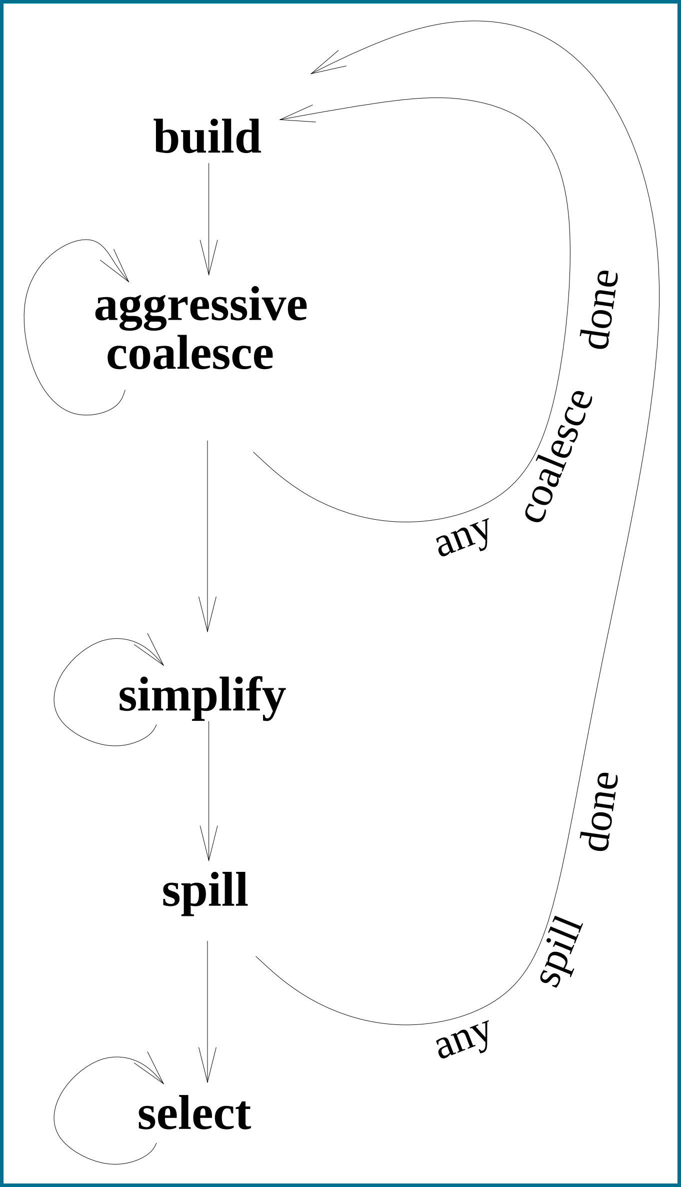
Register allocation by simplification (continued)

4. *Select*: assign colors to nodes
   (a) start with empty graph
   (b) if adding non-spill node there must be a color for it as that was the basis for its removal
   (c) if adding a spill node and no color available (neighbors already K-colored) then mark as an *actual spill*
   (d) repeat select
5. *Start over*: if select has no actual spills then finished, otherwise
   (a) rewrite program to fetch actual spills before each use and store after each definition
   (b) recalculate liveness and repeat

Coalescing

- Can delete a *move* instruction when source s and destination d do not interfere:
  - *coalesce* them into a new node whose edges are the union of those of s and d
- In principle, any pair of non-interfering nodes can be coalesced
  - unfortunately, the union is more constrained and new graph may no longer be K-colorable
  - overly aggressive

Simplification with aggressive coalescing



Conservative coalescing

Apply tests for coalescing that preserve colorability.

Suppose a and b are candidates for coalescing into node ab

Briggs: coalesce only if ab has $< K$ neighbors of *signifi cant* degree $\geq K$

- *simplify* will first remove all insignificant-degree neighbors
- ab will then be adjacent to $< K$ neighbors
- *simplify* can then remove ab

George: coalesce only if all significant-degree neighbors of a already interfere with b

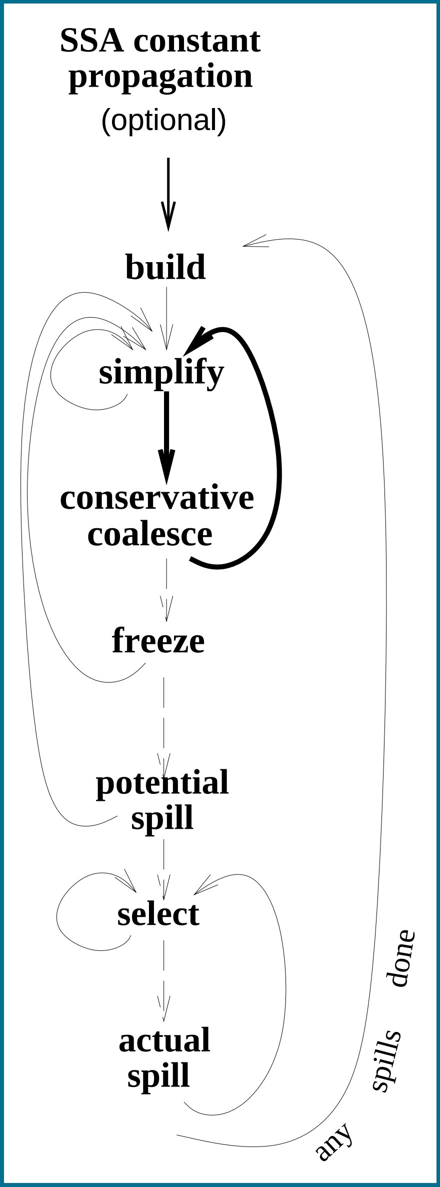
- *simplify* can remove all insignificant-degree neighbors of a
- remaining significant-degree neighbors of a already interfere with b so coalescing does not increase the degree of any node

Iterated register coalescing

Interleave simplifi cation with coalescing to eliminate most moves while without extra spills

1. *Build* interference graph G; distinguish move-related from non-move-related nodes
2. *Simplify*: remove non-move-related nodes of low degree one at a time
3. *Coalesce*: conservatively coalesce move-related nodes
   - remove associated move instruction
   - if resulting node is non-move-related it can now be simplifi ed
   - repeat simplify and coalesce until only signifi cant-degree or uncoalesced moves
4. *Freeze*: if unable to simplify or coalesce
   (a) look for move-related node of low-degree
   (b) freeze its associated moves (give up hope of coalescing them)
   (c) now treat as a non-move-related and resume iteration of simplify and coalesce
5. *Spill*: if no low-degree nodes
   (a) select candidate for spilling
   (b) remove to stack and continue simplifying
6. *Select*: pop stack assigning colors (including actual spills)
7. *Start over*: if select has no actual spills then fi nished, otherwise
   (a) rewrite code to fetch actual spills before each use and store after each defi nition
   (b) recalculate liveness and repeat

Iterated register coalescing



Spilling

- Spills require repeating *build* and *simplify* on the whole program
- To avoid increasing number of spills in future rounds of *build* can simply discard coalescences
- Alternatively, preserve coalescences from before first *potential* spill, discard those after that point
- Move-related spilled temporaries can be aggressively coalesced, since (unlike registers) there is no limit on the number of stack-frame locations

Precolored nodes

Precolored nodes correspond to machine registers (e.g., stack pointer, arguments, return address, return value)

- *select* and *coalesce* can give an ordinary temporary the same color as a precolored register, if they don't interfere
- e.g., argument registers can be reused inside procedures for a temporary
- *simplify*, *freeze* and *spill* cannot be performed on them
- also, precolored nodes interfere with other precolored nodes

So, treat precolored nodes as having infinite degree

This also avoids needing to store large adjacency lists for precolored nodes; coalescing can use the George criterion

Temporary copies of machine registers

Since precolored nodes don’t spill, their live ranges must be kept short:

1. use *move* instructions
2. move callee-save registers to fresh temporaries on procedure entry, and back on exit, spilling between as necessary
3. *register pressure* will spill the fresh temporaries as necessary, otherwise they can be coalesced with their precolored counterpart and the moves deleted

Caller-save and callee-save registers

Variables whose live ranges span calls should go to callee-save registers, otherwise to caller-save

This is easy for graph coloring allocation with spilling

- calls interfere with caller-save registers
- a cross-call variable interferes with all precolored caller-save registers, as well as with the fresh temporaries created for callee-save copies, forcing a spill
- choose nodes with high degree but few uses, to spill the fresh callee-save temporary instead of the cross-call variable
- this makes the original callee-save register available for coloring the cross-call variable

Example

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return [ r1, r3 live out ]
```

- Temporaries are `a`, `b`, `c`, `d`, `e`
- Assume target machine with $K = 3$ registers: `r1`, `r2` (caller-save/argument/resul `r3` (callee-save)
- The code generator has already made arrangements to save `r3` explicitly by copying into temporary `a` and back again

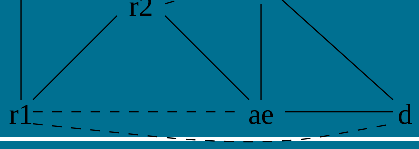
Example (cont.)

- Interference graph:
- No opportunity for *simplify* or *freeze* (all non-precolored nodes have significant degree $\geq K$)
- Any *coalesce* will produce a new node adjacent to $\geq K$ significant-degree nodes
- Must *spill* based on priorities:

Node		uses + defs outside loop		uses + defs inside loop		degree		priority
`a`	(	2	$+10\times$	0	)/	4	=	0.50
`b`	(	1	$+10\times$	1	)/	4	=	2.75
`c`	(	2	$+10\times$	0	)/	6	=	0.33
`d`	(	2	$+10\times$	2	)/	4	=	5.50
`e`	(	1	$+10\times$	3	)/	3	=	10.30

- Node `c` has lowest priority so spill it

Example (cont.)



- Interference graph with `c` removed:

- Only possibility is to *coalesce* `a` and `e`: `ae` will have $< K$ significant-degree neighbors (after coalescing `d` will be low-degree, though high-degree before)

Example (cont.)

- Can now *coalesce* `b` with `r2` (or coalesce `ae` and `r1`):

- *Coalescing* `ae` and `r1` (could also coalesce `d` with `r1`):

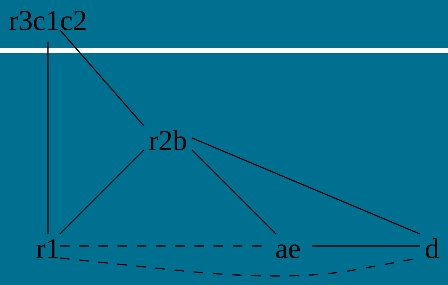
Example (cont.)

- Cannot *coalesce* `r1ae` with `d` because the move is *constrained*: the nodes interfere. Must *simplify* `d`:
- Graph now has only precolored nodes, so pop nodes from stack coloring along the way
  - `d` $\equiv$ `r3`
  - `a`, `b`, `e` have colors by coalescing
  - `c` must spill since no color can be found for it
- Introduce new temporaries `c1` and `c2` for each use/def, add loads before each use and stores after each def

Example (cont.)

```
enter:
  c1 := r3
  M[c_loc] := c1
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  c2 := M[c_loc]
  r3 := c2
  return [ r1, r3 live out ]
```

Example (cont.)



- New interference graph:

- *Coalesce* `c1` with `r3`, then `c2` with `r3`:

- As before, *coalesce* `a` with `e`, then `b` with `r2`:

Example (cont.)

- As before, *coalesce* `ae` with `r1` and *simplify* `d`:
- Pop `d` from stack: select `r3`. All other nodes were coalesced or precolored. So, the coloring is:
  - `a` $\equiv$ `r1`
  - `b` $\equiv$ `r2`
  - `c` $\equiv$ `r3`
  - `d` $\equiv$ `r3`
  - `e` $\equiv$ `r1`

Example (cont.)

- Rewrite the program with this assignment:

```
enter:
  r3 := r3
  M[c_loc] := r3
  r1 := r1
  r2 := r2
  r3 := 0
  r1 := r1
loop:
  r2 := r3 + r2
  r1 := r1 - 1
  if r1 > 0 goto loop
  r1 := r3
  r3 := M[c_loc]
  r3 := r3
  return [ r1, r3 live out ]
```

Example (cont.)

- Delete moves with source and destination the same (coalesced):

```
enter:
  M[c_loc] := r3
  r3 := 0
loop:
  r2 := r3 + r2
  r1 := r1 - 1
  if r1 > 0 goto loop
  r1 := r3
  r3 := M[c_loc]
  return [ r1, r3 live out ]
```

- One uncoalesced move remains

Chapter 9: Activation Records

The procedure abstraction

Separate compilation:

- allows us to build large programs
- keeps compile times reasonable
- requires independent procedures

The linkage convention:

- a social contract
- machine dependent
- division of responsibility

The linkage convention ensures that procedures inherit a valid run-time environment *and* that they restore one for their parents.

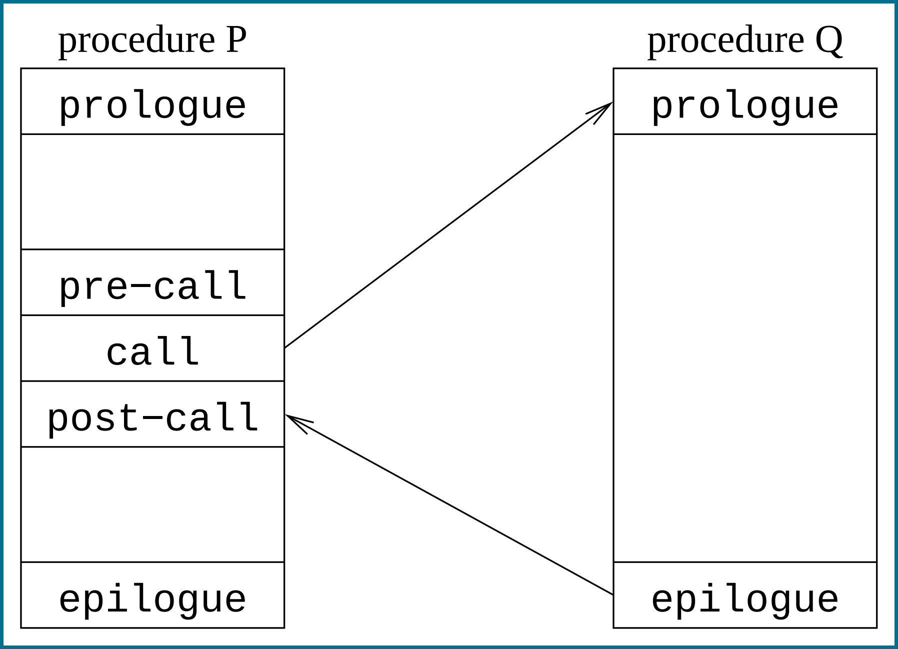
Linkages execute at *run time*

Code to make the linkage is generated at *compile time*

The procedure abstraction

The essentials:

- *on entry*, establish `p`'s environment
- *at a call*, preserve `p`'s environment
- *on exit*, tear down `p`'s environment
- *in between*, addressability and proper lifetimes



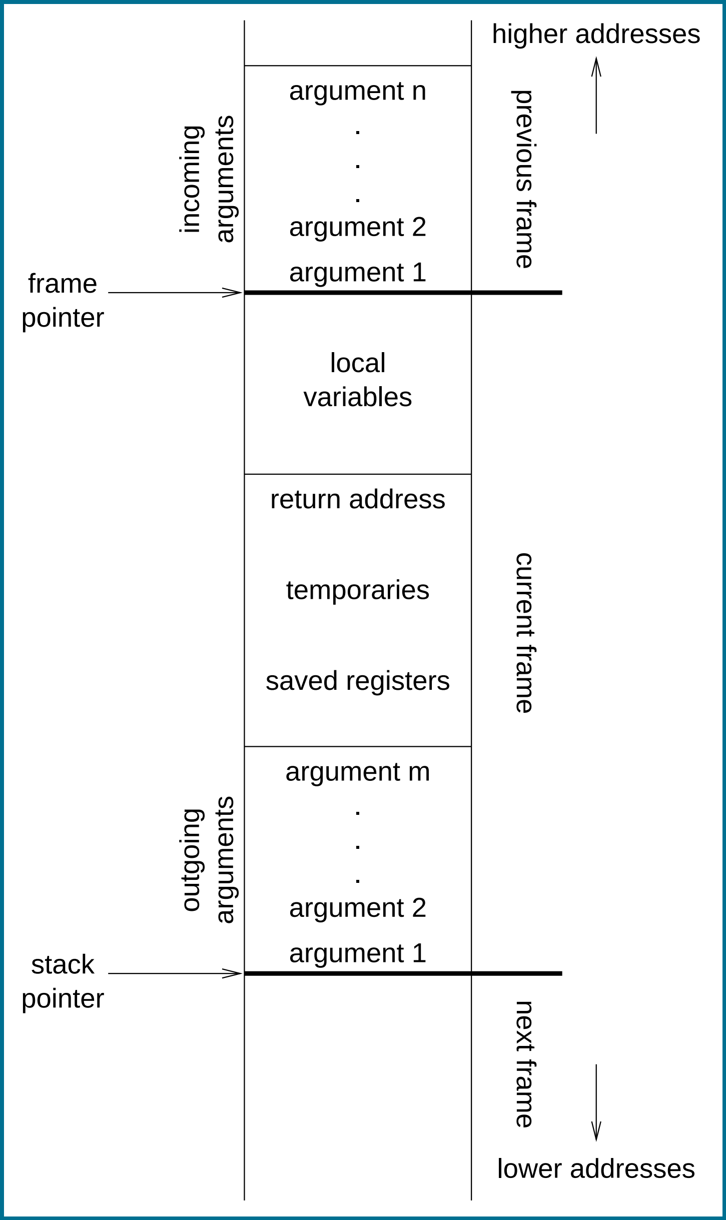
Each system has a *standard linkage*

Procedure linkages

Assume that each procedure activation has an associated *activation record* or *frame* (*at run time*)

Assumptions:

- RISC architecture
- can always expand an allocated block
- locals stored in frame



Procedure linkages

The linkage divides responsibility between *caller* and *callee*

	Caller	Callee
Call	*pre-call*	*prologue*
	1. allocate basic frame 2. evaluate & store params. 3. store return address 4. jump to child	1. save registers, state 2. store FP (dynamic link) 3. set new FP 4. store static link 5. extend basic frame (for local data) 6. initialize locals 7. fall through to code
Return	*post-call*	*epilogue*
	1. copy return value 2. deallocate basic frame 3. restore parameters (if copy out)	1. store return value 2. restore state 3. cut back to basic frame 4. restore parent's FP 5. jump to return address

At compile time, generate the code to do this

At run time, that code manipulates the frame & data areas

Run-time storage organization

To maintain the illusion of procedures, the compiler can adopt some conventions to govern memory use.

Code space

- fixed size
- statically allocated (*link time*)

Data space

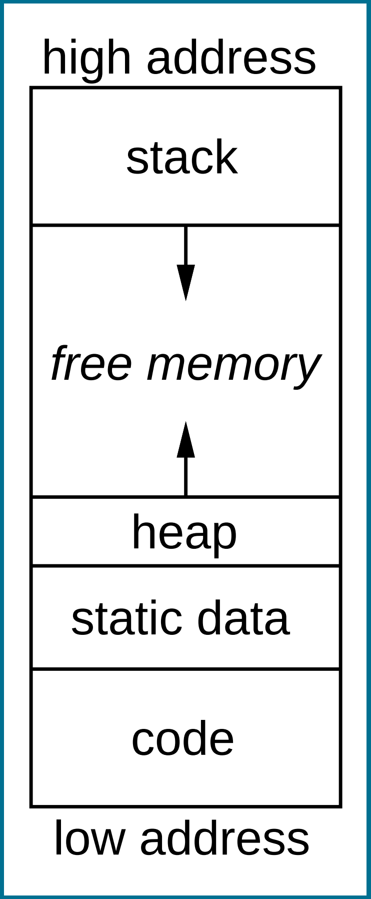
- fixed-sized data may be statically allocated
- variable-sized data must be dynamically allocated
- some data is dynamically allocated in code

Control stack

- dynamic slice of activation tree
- return addresses
- may be implemented in hardware

Run-time storage organization

Typical memory layout



The classical scheme

- allows both stack and heap maximal freedom
- code and static data may be separate or intermingled

Run-time storage organization

Where do local variables go?

When can we allocate them on a stack?

Key issue is lifetime of local names

Downward exposure:

- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:

- can I return a reference to my variables?
- functions that return functions
- continuation-passing style

With only *downward exposure*, the compiler can allocate the frames on the run-time call stack

Storage classes

Each variable must be assigned a storage class (*base address*)

Static variables:

- addresses compiled into code (*relocatable*)
- (*usually*) allocated at compile-time
- limited to fixed size objects
- control access with naming scheme

Global variables:

- almost identical to static variables
- layout may be important (*exposed*)
- naming scheme ensures universal access

Link editor must handle duplicate definitions

Storage classes (*cont.*)

Procedure local variables

Put them on the stack —

- *if* sizes are fixed
- *if* lifetimes are limited
- *if* values are not preserved

Dynamically allocated variables

Must be treated differently —

- call-by-reference, pointers, lead to non-local lifetimes
- (*usually*) an explicit allocation
- explicit or implicit deallocation

Access to non-local data

How does the code find non-local data at *run-time*?

Real globals

- visible *everywhere*
- naming convention gives an address
- initialization requires cooperation

Lexical nesting

- view variables as (*level*,*offset*) pairs (*compile-time*)
- chain of non-local access links
- more expensive to find (*at run-time*)

Access to non-local data

Two important problems arise

- How do we map a name into a (*level,offset*) pair?

  Use a *block-structured symbol table* (remember last lecture?)
  - look up a name, want its most recent declaration
  - declaration may be at current level or any lower level
- Given a (*level,offset*) pair, what's the address?

  Two classic approaches
  - access links (or *static links*)
  - displays

Access to non-local data

To find the value specified by (l, o)

- need current procedure level, k
- $k = l \Rightarrow$ local value
- $k > l \Rightarrow$ find l's activation record
- $k < l$ cannot occur

Maintaining access links: (*static links*)

- calling level $k+1$ procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level $l < k$
  1. find link to level $l-1$ and pass it
  2. its access link will work for lower levels

The display

To improve run-time access costs, use a *display*:

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame
- for level k procedure, need $k-1$ slots

Access with the display

assume a value described by (l, o)

- find slot as `display`$[l]$
- add offset to pointer from slot (`display`$[l][o]$)

“Setting up the basic frame” now includes display manipulation

Calls: Saving and restoring registers

	caller's registers	callee's registers	all registers
callee saves	1	3	5
caller saves	2	4	6

1. Call includes bitmap of caller's registers to be saved/restored
   (best with save/restore instructions to interpret bitmap directly)

2. Caller saves and restores its own registers
   Unstructured returns (e.g., non-local gotos, exceptions) create some problems, since code to restore must be located and executed

3. Backpatch code to save registers used in callee on entry, restore on exit
   e.g., VAX places bitmap in callee's stack frame for use on call/return/non-local goto/exception
   Non-local gotos and exceptions must unwind dynamic chain restoring callee-saved registers

4. Bitmap in callee's stack frame is used by caller to save/restore
   (best with save/restore instructions to interpret bitmap directly)
   Unwind dynamic chain as for 3

5. Easy
   Non-local gotos and exceptions must restore all registers from "outermost callee"

6. Easy (use utility routine to keep calls compact)
   Non-local gotos and exceptions need only restore original registers from caller

Top-left is best: saves fewer registers, compact calling sequences

Call/return

Assuming callee saves:

1. caller pushes space for return value
2. caller pushes SP
3. caller pushes space for:
   return address, static chain, saved registers
4. caller evaluates and pushes actuals onto stack
5. caller sets return address, callee's static chain, performs call

6. callee saves registers in register-save area
7. callee copies by-value arrays/records using addresses passed as ac-
   tuals
8. callee allocates dynamic arrays as needed

9. on return, callee restores saved registers
10. jumps to return address

Caller must allocate much of stack frame, because it computes the actual parameters

Alternative is to put actuals below callee's stack frame in caller's: common when hardware supports stack management (e.g., VAX)

MIPS procedure call convention

Registers:

Number	Name	Usage
0	`zero`	Constant 0
1	at	Reserved for assembler
2, 3	v0, v1	Expression evaluation, scalar function results
4–7	a0–a3	first 4 scalar arguments
8–15	t0–t7	Temporaries, caller-saved; caller must save to preserve across calls
16–23	s0–s7	Callee-saved; must be preserved across calls
24, 25	t8, t9	Temporaries, caller-saved; caller must save to preserve across calls
26, 27	k0, k1	Reserved for OS kernel
28	gp	Pointer to global area
29	sp	Stack pointer
30	s8 (fp)	Callee-saved; must be preserved across calls
31	ra	Expression evaluation, pass return address in calls

MIPS procedure call convention

Philosophy:

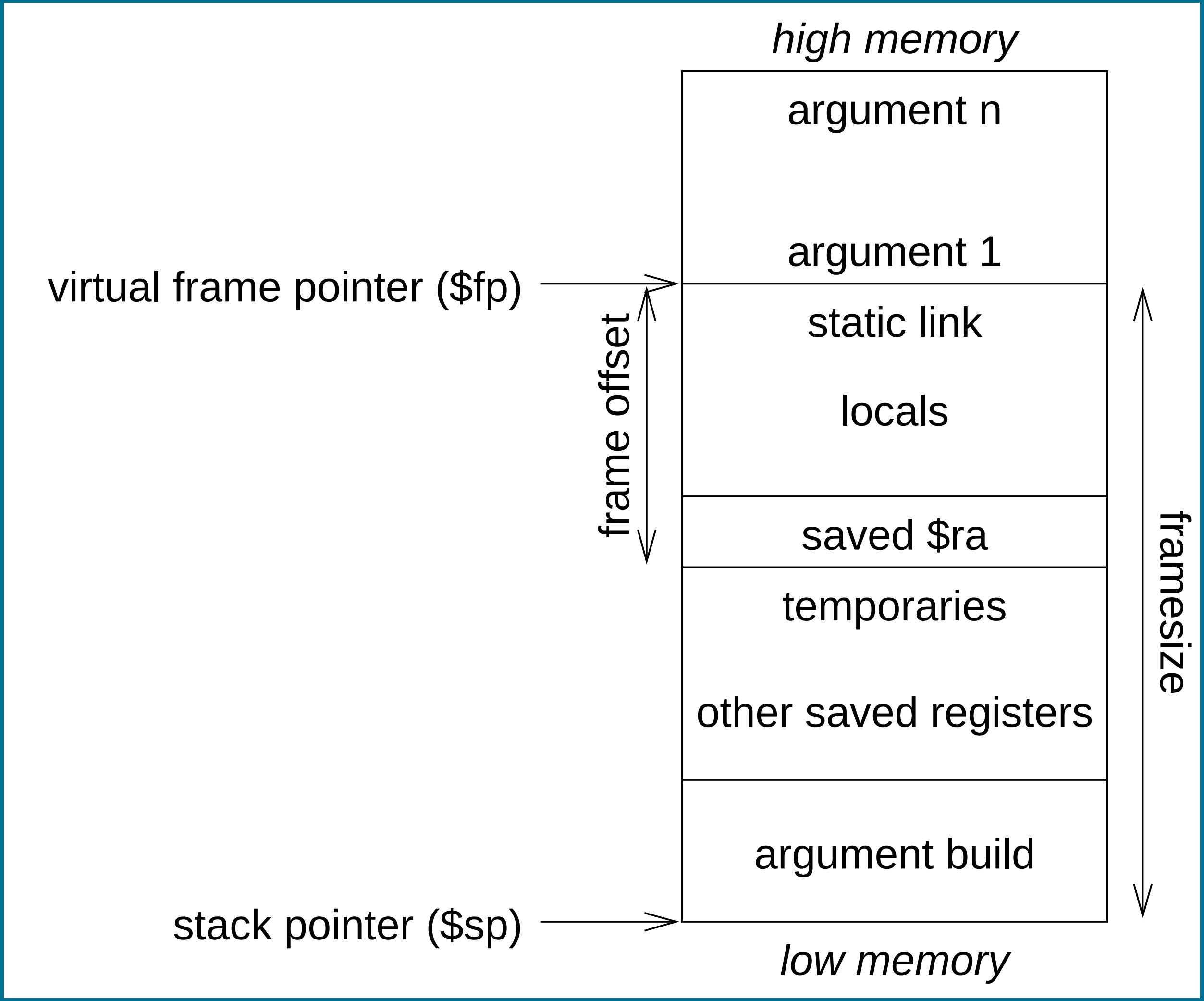
> Use full, general calling sequence only when necessary; omit portions of it where possible (e.g., avoid using fp register whenever possible)

Classify routines as:

- non-leaf routines: routines that call other routines
- leaf routines: routines that do not themselves call other routines
  - leaf routines that require stack storage for locals
  - leaf routines that do not require stack storage for locals

MIPS procedure call convention

The stack frame



MIPS procedure call convention

Pre-call:

1. Pass arguments: use registers a0 . . . a3; remaining arguments are pushed on the stack along with save space for a0 . . . a3
2. Save caller-saved registers if necessary
3. Execute a `jal` instruction: jumps to target address (callee's first instruction), saves return address in register ra

MIPS procedure call convention

Prologue:

1. Leaf procedures that use the stack and non-leaf procedures:
   (a) Allocate all stack space needed by routine:
      - local variables
      - saved registers
      - sufficient space for arguments to routines called by this routine

      ```
      subu $sp,framesize
      ```

   (b) Save registers (ra, etc.)
      e.g.,

      ```
      sw $31,framesize+frameoffset($sp)
      sw $17,framesize+frameoffset-4($sp)
      sw $16,framesize+frameoffset-8($sp)
      ```

      where `framesize` and `frameoffset` (usually negative) are compile-time constants
2. Emit code for routine

MIPS procedure call convention

Epilogue:

1. Copy return values into result registers (if not already there)
2. Restore saved registers

   ```
   lw reg,framesize+frameoffset-N($sp)
   ```

3. Get return address

   ```
   lw $31,framesize+frameoffset($sp)
   ```

4. Clean up stack

   ```
   addu $sp,framesize
   ```

5. Return

   ```
   j $31
   ```