CS 352 Compilers: Principles and Practice

1. Introduction 2
2. Lexical analysis 31
3. LL parsing 58
4. LR parsing 110
5. JavaCC and JTB 127
6. Semantic analysis 150
7. Translation and simplification 165
8. Liveness analysis and register allocation 185
9. Activation Records 216

Chapter 1: Introduction

Things to do

- make sure you have a working mentor account
- start brushing up on Java
- review Java development tools
- find http://www.cs.purdue.edu/homes/palsberg/cs352/F00/index.html
- add yourself to the course mailing list by writing (on a CS computer)
  
  mailer add me to cs352

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

<table>
<thead>
<tr>
<th>artificial intelligence</th>
<th>greedy algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithms</td>
<td>learning algorithms</td>
</tr>
<tr>
<td>graph algorithms</td>
<td>union-find algorithms</td>
</tr>
<tr>
<td>theory</td>
<td>DFAs for scanning parser generators lattice theory for analysis</td>
</tr>
<tr>
<td>systems</td>
<td>allocation and naming locality synchronization</td>
</tr>
<tr>
<td>architecture</td>
<td>pipeline management hierarchy management instruction set use</td>
</tr>
</tbody>
</table>

Inside a compiler, all these things come together

Isn’t it a solved problem?

Machines are constantly changing

Changes in architecture ⇒ 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 ⇒ 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**

context
code

compiler

machine
code

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**

source
code

front
end

IR

back
end

machine
code

errors

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 ⇒ better code

**A fallacy**

FORTRAN
code

C++
code

CLU
code

Smalltalk
code

front
end

back
end

target1

target2

target3

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

Scanner:

- maps characters into tokens – the basic unit of syntax
  \( x = x + y; \) becomes
  \(<\text{id}, x> + <\text{id}, x> + <\text{id}, y> ;\)
- 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)

Context-free syntax is specified with a grammar

\[
<\text{sheep noise}> ::= \text{baa} \\
| \text{baa} <\text{sheep noise}>
\]

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

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

<table>
<thead>
<tr>
<th>Prod'n.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;term&gt; ::= number</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; ::= id</td>
</tr>
<tr>
<td>5</td>
<td>&lt;op&gt; ::= +</td>
</tr>
<tr>
<td>6</td>
<td>&lt;op&gt; ::= -</td>
</tr>
</tbody>
</table>

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

$S = <goal>$

$T = number, id, +, -$  

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

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

17

Front end

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

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

18

19

20
Back end

IR → instruction selection → register allocation → machine code

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

愠mation has been less successful here

Back end

IR → instruction selection → register allocation → machine code

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

IR → instruction selection → register allocation → machine code

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

source code → front end → IR → middle end → IR → back end → machine code

Code Improvement

- Analyzes and changes IR
- Goal is to reduce runtime
- Must preserve values
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

The Tiger compiler phases

<table>
<thead>
<tr>
<th>Pass</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lex</td>
<td>Break source file into individual words, or tokens</td>
</tr>
<tr>
<td>Parse</td>
<td>Analyse the phrase structure of program</td>
</tr>
<tr>
<td>Parsing Actions</td>
<td>Build a piece of abstract syntax tree for each phrase</td>
</tr>
<tr>
<td>Semantic Analysis</td>
<td>Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase</td>
</tr>
<tr>
<td>Frame Layout</td>
<td>Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way</td>
</tr>
<tr>
<td>Canonicalize</td>
<td>Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases</td>
</tr>
<tr>
<td>Instruction Selection</td>
<td>Group IR-tree nodes into clumps that correspond to actions of target-machine instructions</td>
</tr>
<tr>
<td>Control Flow Analysis</td>
<td>Analyse sequence of instructions into control flow graph showing all possible flows of control program might follow when it runs</td>
</tr>
<tr>
<td>Data Flow Analysis</td>
<td>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</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Choose registers for variables and temporary values; variables not simultaneously live can share same register</td>
</tr>
<tr>
<td>Code Emission</td>
<td>Replace temporary names in each machine instruction with registers</td>
</tr>
</tbody>
</table>

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

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

  \[
  \begin{align*}
  Stm & \rightarrow Stm \; ; \; Stm \quad \text{CompoundStm} \\
  Stm & \rightarrow \text{id} \; := \; \text{Exp} \quad \text{AssignStm} \\
  Stm & \rightarrow \text{print} \; ( \; \text{ExpList} \; ) \quad \text{PrintStm} \\
  Exp & \rightarrow \text{id} \quad \text{IdExp} \\
  Exp & \rightarrow \text{num} \quad \text{NumExp} \\
  Exp & \rightarrow \text{Exp} \; \text{Binop} \; \text{Exp} \quad \text{OpExp} \\
  Exp & \rightarrow ( \; \text{Stm} \; , \; \text{Exp} \; ) \quad \text{EqExp} \\
  \text{ExpList} & \rightarrow \text{Exp} \; , \; \text{ExpList} \quad \text{PairExpList} \\
  \text{ExpList} & \rightarrow \text{Exp} \quad \text{LastExpList} \\
  \text{Binop} & \rightarrow + \quad \text{Plus} \\
  \text{Binop} & \rightarrow - \quad \text{Minus} \\
  \text{Binop} & \rightarrow \times \quad \text{Times} \\
  \text{Binop} & \rightarrow \div \quad \text{Div} \\
  \end{align*}
\]

- e.g., \( a := 5 + 3, \ b := (\text{print}(a, a - 1), 10 \times a) \): \text{print}(b)

prints:

\[
\begin{align*}
8 & 7 \\
80 &
\end{align*}
\]
Chapter 2: Lexical Analysis

Tree representation

\[
a := 5 + 3; \quad b := (\text{print}(a,a-1), 10 \times a); \quad \text{print}(b)
\]

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

Java classes for trees

```java
abstract class Stmt {
    // ...
    class CompoundStmt extends Stmt {
        String stml, stmr;
        CompoundStmt(Stmt stl, Stmt st2) {
            stml = stl; stmr = st2;
        }
    }

    class AssignStmt extends Stmt {
        String id; Exp exp;
        AssignStmt(String i, Exp e) {
            id = i; expr = e;
        }
    }

    class PrintStmt extends Stmt {
        ExpList expr;
        PrintStmt(ExpList e) {
            expr = 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
        OpExp(Exp l, int o, Exp r) {
            left = l; oper = o; right = r;
        }
    }

    class ReadExp extends Exp {
        Exp readExp;
        ReadExp(Exp s) {
            Exp read = s;
        }
    }

    abstract class List {
        class PairExpList extends List {
            Exp head; ExpList tail;
            public PairExpList(Exp h, ExpList t) {
                head = h; tail = t;
            }
        }
    }

    class LastExpList extends List {
        Exp head;
        public LastExpList(Exp h) {head = h;}
    }
}
```

Scanner

- maps characters into tokens – the basic unit of syntax
- becomes
  \[
  \langle \text{id}, x \rangle \rightarrow \langle \text{id}, x \rangle + \langle \text{id}, y \rangle;
  \]
- 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`)

Copyright ©2000 by Antony L. Hosking. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@ics.purdue.edu.
Specifying patterns

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

Some parts are easy:

- **white space**
  
  `<WS>` ::= `<WS>` `' ' | `<WS>` `'\t'` | `<WS>` `'\r'`

- **keywords and operators**
  specified as literal patterns: `do`, `end`

- **comments**
  opening and closing delimiters: `/* ... */`

Other parts are much harder:

- **identifiers**
  alphabetic followed by alphanumeric (`a`, `b`, `c`, ...)

- **numbers**
  integers: 0 or digit from 1-9 followed by digits from 0-9
  decimals: integer `.` `digit` digits from 0-9
  reals: (integer or decimal) `+` `digit` `digit` `digit`

We need a powerful notation to specify these patterns

### Operations on languages

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>union of <code>L</code> and <code>M</code> written <code>L ∪ M</code></td>
<td>`L ∪ M = { x</td>
</tr>
<tr>
<td>concatenation of <code>L</code> and <code>M</code> written <code>LM</code></td>
<td>`LM = { x</td>
</tr>
<tr>
<td>Kleene closure of <code>L</code> written <code>L*</code></td>
<td><code>L* = ∪_{i=0}^{∞} L^i</code></td>
</tr>
<tr>
<td>positive closure of <code>L</code> written <code>L+</code></td>
<td><code>L+ = ∪_{i=1}^{∞} L^i</code></td>
</tr>
</tbody>
</table>

### 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 `{ ε }`
2. if `a` ∈ `Σ`, 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 | s }` is a RE denoting `L(r) ∪ L(s)`
   - `{ r }{ s }` is a RE denoting `L(r)L(s)`
   - `{ r }{ r }` is a RE denoting `L(r) +` (positive closure)

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**
  - \( \text{letter} \rightarrow \{a, b, c, \ldots, z, A, B, C, \ldots, Z\} \)
  - \( \text{digit} \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
  - \( \text{id} \rightarrow \text{letter} \ (\text{letter} \mid \text{digit})^* \)

- **numbers**
  - \( \text{integer} \rightarrow \{+, -, 0, 1, 2, 3, \ldots, 9\} \ (\text{digit})^* \)
  - \( \text{decimal} \rightarrow \text{integer} \ . \ (\text{digit})^* \)
  - \( \text{real} \rightarrow \{\text{integer} \mid \text{decimal}\} \ (\{+\mid-\}\text{digit}^*) \)
  - \( \text{complex} \rightarrow \{(+\text{real} \mid \text{real}^*)^* \}

**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

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r \epsilon = r \epsilon )</td>
<td>is commutative</td>
</tr>
<tr>
<td>( r[[r]] = r[[r]] )</td>
<td>is associative</td>
</tr>
<tr>
<td>( (rs)t = r(st) )</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>( r(sr) = rsrt )</td>
<td>concatenation distributes over (</td>
</tr>
<tr>
<td>( r \epsilon = r \epsilon = r \epsilon )</td>
<td>( \epsilon ) is the identity for concatenation</td>
</tr>
</tbody>
</table>

\[ r^* = [r] \cup [r]^2 \cup [r]^3 \cup \ldots \] relation between \( \epsilon \) and \( r \)

\[ r^{2*} = r^* \] \( \epsilon \) is idempotent

### Recognizers

From a regular expression we can construct a

**deterministic finite automaton (DFA)**

#### Recognizer for identifier:

![Deterministic finite automaton (DFA)](image)

**identifier**

- \( \text{letter} \rightarrow \{a, b, c, \ldots, z, A, B, C, \ldots, Z\} \)
- \( \text{digit} \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
- \( \text{id} \rightarrow \text{letter} \ (\text{letter} \mid \text{digit})^* \)
Code for the recognizer

```c
char ← next_char();
state ← 0;     // code for state 0 */
done ← false;  // 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

<table>
<thead>
<tr>
<th>char_class</th>
<th>a–z</th>
<th>A–Z</th>
<th>0–9</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>digit</td>
<td>3</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>other</td>
<td>3</td>
<td>2</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

To change languages, we can just change tables

Grammar 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. $A → αA$
2. $A → α$

where $A$ is any non-terminal and $α$ is any terminal symbol

These are also called type 3 grammars (Chomsky)

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)
More regular languages

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

![Diagram of a finite automaton](image)

The RE is $(00|11)*(0|110|11)*(0|10|00|11)*$.

More regular expressions

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

State $s_0$ has multiple transitions on $a$.

$\Rightarrow$ nondeterministic finite automaton

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_0,s_1$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$-$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

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

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 $\Sigma$ (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 $\epsilon$-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 $x$. 
NFA to DFA using the subset construction: example 1

Constructing a DFA from a regular expression

RE to NFA

RE to NFA: example
NFA to DFA: the subset construction

Input: NFA $N$
Output: A DFA $D$ with states $D$states and transitions $D$trans 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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$-closure($s$)</td>
<td>set of NFA states reachable from NFA state $s$ on $\epsilon$-transitions alone</td>
</tr>
<tr>
<td>$\epsilon$-closure($T$)</td>
<td>set of NFA states reachable from some NFA state in $T$ on $\epsilon$-transitions alone</td>
</tr>
<tr>
<td>move($T$, $a$)</td>
<td>set of NFA states to which there is a transition on input symbol $a$ from some NFA state in $T$</td>
</tr>
</tbody>
</table>

1. Add state $T = \epsilon$-closure($s$) unmarked to $D$states
2. Mark $T$
3. For each input symbol $a$:
   - $U = \epsilon$-closure(move($T$, $a$))
   - If $U \notin D$states then add $U$ to $D$states unmarked
   - $D$trans[$T$, $a$] = $U$
4. $\epsilon$-closure($s$) is the start state of $D$
5. $A$ state of $D$ is accepting if it contains at least one accepting state in $N$

Limits of regular languages

Not all languages are regular

One cannot construct DFAs to recognize these languages:

- $L = \{ p^4 q^3 \}$
- $L = \{ w w^r \mid w \in \Sigma^* \}$

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
  - $\{ x \mid (01)^* x \}$
- 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 else; else else then;
- **significant blanks**
  - FORTRAN and Algol68 ignore blanks
  - do 10 i = 1, 25
  - do 10 i = 1, 26
- **string constants**
  - special characters in strings
  - newline, tab, quote, comment delimiter
- **finite 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
Chapter 3: LL Parsing

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

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

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

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

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

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

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

Syntax analysis

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

Example:

1. $\{\text{goal}\} ::= \{\text{expr}\}$
2. $\{\text{expr}\} ::= \{\text{expr}\}/\{\text{op}\}/\{\text{expr}\}$
3. $\{\text{op}\} ::= \{\text{num}\}$
4. $\{\text{op}\} ::= \{+\}$
5. $\{\text{op}\} ::= \{-\}$
6. $\{\text{op}\} ::= \{\text{num}\}$
7. $\{\text{op}\} ::= \{/\}$

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 underline
3. productions as in the example

Derivations

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

Using our example CFG:

\[
\begin{align*}
\{\text{goal}\} & \Rightarrow \{\text{expr}\} \\
\{\text{expr}\} & \Rightarrow \{\text{expr}\}/\{\text{op}\}/\{\text{expr}\} \\
\{\text{op}\} & \Rightarrow \{\text{num}\} \\
\{\text{op}\} & \Rightarrow \{+\} \\
\{\text{op}\} & \Rightarrow \{-\} \\
\{\text{op}\} & \Rightarrow \{\text{num}\} \\
\{\text{op}\} & \Rightarrow \{/\}
\end{align*}
\]

We have derived the sentence $x + 2 \times y$.

We denote this $\{\text{goal}\} \Rightarrow^* \{\text{id}\} + \{\text{num}\} \times \{\text{id}\}$.

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

The process of discovering a derivation is called parsing.

Scanning vs. parsing

Where do we draw the line?

- $\text{term} ::= [a-z\alpha-x][0-9]^*$
- $\text{op} ::= [\{+\} | [-] | /]$
- $\text{expr} ::= \{\text{term}\}\{\text{op}\}\{\text{term}\}$

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

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 \( x + 2 \times y \):

\[
\begin{align*}
\text{(goal)} & \Rightarrow (\text{expr}) \\
& \Rightarrow (\text{expr}) \{\text{op}\} (\text{expr}) \\
& \Rightarrow (\text{expr}) \{\text{op}\} (\text{id}, y) \\
& \Rightarrow (\text{expr}) \times (\text{id}, y) \\
& \Rightarrow (\text{expr}) \{\text{op}\} (\text{num}, 2) \times (\text{id}, y) \\
& \Rightarrow (\text{expr}) + (\text{num}, 2) \times (\text{id}, y) \\
& \Rightarrow (\text{id}, x) + (\text{num}, 2) \times (\text{id}, y)
\end{align*}
\]

Again, \((\text{goal}) \Rightarrow * \text{id} + \text{num} \times \text{id}\).

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 \[
\text{(goal)}::=\text{(expr)}
\]
2 \[
\text{(expr)}::=\text{(expr)}\{\text{op}\}\text{term}
\]
3 \[
\text{term}::=\text{term}\{\text{op}\}\text{factor}
\]
4 \[
\text{factor}::=\text{num} \quad \text{or} \quad \text{id}
\]
5 \[
\text{term}::=\text{term}\{\text{op}\}\text{factor}
\]
6 \[
\text{factor}::=\text{num} \quad \text{or} \quad \text{id}
\]

This grammar enforces a precedence on the derivation:

- terms must be derived from expressions
- forces the “correct” tree
**Precedence**

Now, for the string $x + 2 * y$:

\[
\begin{align*}
\text{(goal)} & \Rightarrow \text{(expr)} \\
\text{(expr)} & \Rightarrow (\text{term}) \\
\text{(term)} & \Rightarrow (\text{term}) + (\text{factor}) \\
\text{(factor)} & \Rightarrow (\text{term}) * (\text{id} \cdot y) \\
\text{(term)} & \Rightarrow \text{(num,2)} * (\text{id} \cdot y) \\
\text{(factor)} & \Rightarrow \text{(num,2)} * (\text{id} \cdot y) \\
\text{(id, x)} & \Rightarrow \text{(num,2)} * (\text{id} \cdot y)
\end{align*}
\]

Again, \( \text{(goal)} \Rightarrow \text{id + num + id} \), but this time, we build the desired tree.

**Ambiguity**

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

Example:

\[
\begin{align*}
\text{(stmt)} & \ ::= \text{if (expr) then (stmt)} \\
& \quad \text{if (expr) then (stmt) else (stmt)} \\
& \quad \text{other stats}
\end{align*}
\]

Consider deriving the sentential form:

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]

It has two derivations.

This ambiguity is purely grammatical.

It is a context-free ambiguity.

**Tree walk evaluation computes** $x + (2 * y)$

**Ambiguity**

May be able to eliminate ambiguities by rearranging the grammar:

\[
\begin{align*}
\text{(stmt)} & \ ::= \text{[matched]} \\
\text{(matched)} & \ ::= \text{if (expr) then (matched) else (matched)} \\
\text{(unmatched)} & \ ::= \text{if (expr) then (stmt) else (unmatched)}
\end{align*}
\]

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

![Diagram of parsing process]

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 prefixes)
- 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 \( \alpha \)
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 \( V_\alpha \))

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:

1: goal ::= (expr)
2: (expr) ::= (expr) + (term)
3: (term) ::= (expr) - (term)
4: (term) ::= (term) * (factor)
5: (factor) ::= (term) / (factor)
6: (factor) ::= num
7: (factor) ::= id

Consider the input string x - 2*y

Example

Another possible parse for x - 2*y

<table>
<thead>
<tr>
<th>Prod'n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(goal)</td>
<td>x - 2*y</td>
</tr>
<tr>
<td>2</td>
<td>(expr) + (term)</td>
<td>x - 2*y</td>
</tr>
<tr>
<td>2</td>
<td>(expr) + (term) + (term)</td>
<td>x - 2*y</td>
</tr>
<tr>
<td>2</td>
<td>(expr) + (term) + ...</td>
<td>x - 2*y</td>
</tr>
</tbody>
</table>

Our simple expression grammar is left-recursive

Left-recursion

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

Formally, a grammar is left-recursive if

\[ \exists A \in V_0 \text{ such that } A \Rightarrow^+ A\alpha \text{ for some string } \alpha \]

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!)
Eliminating left-recursion

To remove left-recursion, we can transform the grammar

Consider the grammar fragment:

\[ \text{foo} \rightarrow \text{foo} \alpha \]
\[ \text{where } \alpha \text{ do not start with foo} \]

We can rewrite this as:

\[ \text{foo} \rightarrow \beta \]
\[ \text{bar} \rightarrow \alpha \]
\[ \text{bar} \rightarrow \epsilon \]

where \( \text{bar} \) is a new non-terminal

This fragment contains no left-recursion

Example

Our expression grammar contains two cases of left-recursion

\[
\begin{align*}
\text{expr} &::= (\text{expr}) + \text{term} \\
& | (\text{term}) - (\text{term}) \\
\text{term} &::= (\text{factor}) \ast (\text{term}) \\
& | (\text{factor}) / (\text{term}) \\
\text{factor} &::= \text{num} \\
& | \text{id}
\end{align*}
\]

Applying the transformation gives

\[
\begin{align*}
\text{expr} &::= (\text{term}) (\text{expr'}) \\
\text{expr'} &::= + (\text{term}) (\text{expr'}) \\
& | \epsilon \\
\text{term} &::= (\text{factor}) (\text{term}) \\
\text{term'} &::= \ast (\text{factor}) (\text{term}) \\
& | \epsilon \\
& | / (\text{factor}) (\text{term})
\end{align*}
\]

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

\[
\begin{align*}
\text{goal} &::= \text{expr} \\
\text{expr} &::= (\text{term}) + (\text{expr}) \\
& | (\text{term}) - (\text{expr}) \\
\text{term} &::= (\text{factor}) \ast (\text{term}) \\
& | (\text{factor}) / (\text{term}) \\
\text{factor} &::= \text{num} \\
& | \text{id}
\end{align*}
\]

It is
- right-recursive
- free of \( \epsilon \) productions

Unfortunately, it generates different associativity

Same syntax, different meaning

Example

Our long-suffering expression grammar:

\[
\begin{align*}
\text{goal} &::= \text{expr} \\
\text{expr} &::= (\text{term}) (\text{expr'}) \\
\text{expr'} &::= + (\text{term}) (\text{expr'}) \\
\text{term} &::= (\text{factor}) (\text{term}) \\
\text{term'} &::= \ast (\text{factor}) (\text{term}) \\
\text{factor} &::= \text{num} \\
& | \text{id}
\end{align*}
\]

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):** Left to right scan, left-most derivation, 1-token lookahead; and

**LR(1):** Left to right scan, right-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 $\text{FIRST}(\alpha)$ as the set of tokens that appear first in some string derived from $\alpha$.

That is, for some $w \in V_+^*$, $w \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* w$.

**Key property:**

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

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

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 $\alpha$ 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

$$A \rightarrow A'$$

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

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:

<table>
<thead>
<tr>
<th></th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[goal]</td>
</tr>
<tr>
<td>2</td>
<td>(expr)</td>
</tr>
<tr>
<td>3</td>
<td>(term)</td>
</tr>
<tr>
<td>4</td>
<td>(factor)</td>
</tr>
<tr>
<td>5</td>
<td>(factor)</td>
</tr>
<tr>
<td>6</td>
<td>num</td>
</tr>
<tr>
<td>7</td>
<td>id</td>
</tr>
</tbody>
</table>

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

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

This grammar fails the test.

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

There are two nonterminals that must be left factored:

\[
\begin{align*}
\text{(expr)} & \quad ::= \text{(term)} + \text{(expr)} \\
& \quad | \text{(term)} - \text{(expr)} \\
& \quad | \text{(term)} \\
\text{(term)} & \quad ::= \text{(factor)} \ast \text{(term)} \\
& \quad | \text{(factor)} / \text{(term)} \\
& \quad | \text{(factor)} \\
\end{align*}
\]

Applying the transformation gives us:

\[
\begin{align*}
\text{(expr)} & \quad ::= \text{(term)} \text{(expr')}
\text{(expr')} & \quad ::= + \text{(expr)} \\
& \quad | - \text{(expr)} \\
& \quad | \epsilon \\
\text{(term)} & \quad ::= \text{(factor)} \text{(term')} \\
\text{(term')} & \quad ::= \ast \text{(term)} \\
& \quad | / \text{(term)} \\
& \quad | \epsilon \\
\end{align*}
\]

Example

Substituting back into the grammar yields

\[
\begin{align*}
\text{(goal)} & \quad ::= \text{(expr)} \\
\text{(expr)} & \quad ::= \text{(term)} \text{(expr')} \\
\text{(expr')} & \quad ::= + \text{(expr)} \\
& \quad | - \text{(expr)} \\
& \quad | \epsilon \\
\text{(term)} & \quad ::= \text{(factor)} \text{(term')} \\
\text{(term')} & \quad ::= \ast \text{(term)} \\
& \quad | / \text{(term)} \\
& \quad | \epsilon \\
\end{align*}
\]

Now, selection requires only a single token lookahead.

Note: This grammar is still right-associative.

Back to left-recursion elimination

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

\[
\begin{align*}
\text{if } & \exists A \rightarrow \alpha \gamma \text{ then replace all of the } A \text{ productions } \\
& \gamma \rightarrow \alpha \delta | \eta | \ldots | \gamma \\
\text{with} \\
& A \rightarrow N \alpha' \\
& \gamma \rightarrow \beta | \ldots | \gamma \\
& \alpha' \rightarrow \alpha \alpha' | \epsilon \quad \text{where } N \text{ and } \alpha' \text{ are new productions.}
\end{align*}
\]

Repeat until there are no left-recursive productions.

The next symbol determined each choice correctly.
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:

\[ \{a^n b^n | n \geq 1\} \cup \{a^n b^{2m} | m \geq 1\} \]

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;
```

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

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

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

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

What we need now is a parsing table $M$.

Our expression grammar: Its parse table:

<table>
<thead>
<tr>
<th></th>
<th>(goal)</th>
<th>(expr)</th>
<th>(term)</th>
<th>(expr')</th>
<th>(term')</th>
<th>(factor)</th>
<th>id</th>
<th>num</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(goal)</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(expr)</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(term)</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(expr')</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(term')</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(factor)</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 We use $\$\$ to represent EOF

FIRST

For a string of grammar symbols $\alpha$, define FIRST($\alpha$) as:

- the set of terminal symbols that begin strings derived from $\alpha$:
  \[ \{ a \in V_T | \alpha \Rightarrow^* a \} \]
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in$ FIRST($\alpha$)

FIRST($\alpha$) contains the set of tokens valid in the initial position in $\alpha$

To build FIRST($X$):

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

Repeat until no more additions can be made.

FOLLOW

For a non-terminal $A$, define 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 FOLLOW($A$):

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

Repeat until no more additions can be made

LL(1) grammars

Previous definition

A grammar $G$ is LL(1) if, for all non-terminals $A$, each distinct pair of productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition FIRST($\beta$) \cap FIRST($\gamma$) = \{\}

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

Revised definition

A grammar $G$ is LL(1) if, for each set of productions $A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_m$:

1. FIRST($\alpha_1$), FIRST($\alpha_2$), \ldots, FIRST($\alpha_m$) are all pairwise disjoint
2. If $\alpha \Rightarrow^* \varepsilon$ then FIRST($\alpha_j$) \cap FOLLOW($\alpha_k$) = $\emptyset$ $\forall 1 \leq j \leq n, i \neq j$

If $G$ is $\varepsilon$-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

\[ S \rightarrow aS \mid a \]

is not LL(1) because \( \text{FIRST}(aS) = \text{FIRST}(a) - \{a\} \)

\[ S \rightarrow aS' \]

\[ S' \rightarrow aS' \mid \epsilon \]

accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar \( G \)

Output: Parsing table \( M \)

Method:

1. ∀ productions \( A \rightarrow \alpha \):
   (a) ∀ \( a \in \text{FIRST}(\alpha) \), add \( A \rightarrow \alpha \) to \( M[\alpha, a] \)
   (b) If \( \epsilon \in \text{FIRST}(\alpha) \):
      i. ∀ \( a \in \text{FOLLOW}(A) \), add \( A \rightarrow \alpha \) to \( M[\alpha, a] \)
      ii. If \( S \in \text{FOLLOW}(A) \) then add \( A \rightarrow \alpha \) to \( M[\alpha, S] \)

2. Set each undefined entry of \( M \) to \( \epsilon \text{error} \)

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

Note: recall \( a, b \in V \), so \( a, b \neq \epsilon \)

Example

Our long-suffering expression grammar:

\[
\begin{align*}
S &\rightarrow E \\
E &\rightarrow TE' \\
E' &\rightarrow +E \mid -E \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow \ast T \mid \epsilon \\
F &\rightarrow \text{id} \mid \text{num}
\end{align*}
\]

<table>
<thead>
<tr>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \ast \text{num} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{num} )</td>
</tr>
</tbody>
</table>

A grammar that is not LL(1)

\[
\begin{align*}
\langle \text{stmt} \rangle &\ ::= \text{if (expr) then (stmt) | if (expr) then (stmt) else (stmt) | ...} \\
\langle \text{stmt}' \rangle &\ ::= \text{else (stmt) | } \epsilon
\end{align*}
\]

Left-factored:

\[
\begin{align*}
\langle \text{stmt} \rangle &\ ::= \text{if (expr) then (stmt) | (stmt') | ...} \\
\langle \text{stmt}' \rangle &\ ::= \text{else (stmt) | } \epsilon
\end{align*}
\]

Now, \( \text{FIRST}(\langle \text{stmt}' \rangle) = \{ \epsilon, \text{else} \} \)

Also, \( \text{FOLLOW}(\langle \text{stmt}' \rangle) = \{ \text{else}, \text{S} \} \)

But, \( \text{FIRST}(\langle \text{stmt}' \rangle) \cap \text{FOLLOW}(\langle \text{stmt}' \rangle) = \{ \text{else} \} \neq \emptyset \)

On seeing \( \text{else} \), conflict between choosing

\[
\langle \text{stmt}' \rangle \ ::= \text{else (stmt) and (stmt') ::= } \epsilon
\]

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

The fix:

Put priority on \( (\text{stmt}') ::= \text{else (stmt) to associate else with closest previous then.} \)
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 SYNCH(A) is found

Building SYNCH:

1. \( \# \in \text{FOLLOW}(A) \Rightarrow \# \in \text{SYNCH}(A) \)
2. place keywords that start statements in SYNCH(A)
3. add symbols in FIRST(A) to 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., SYNCH(\( A \)) \( \rightarrow \) \( V \_1 \rightarrow \{ \# \} \))

Chapter 4: LR Parsing

Some definitions

Recall

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

- If \( \alpha \in V \_1^* \), then \( \alpha \) 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

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow ABc \\
B & \rightarrow b \\
\end{align*}
\]

and the input string abbcde

<table>
<thead>
<tr>
<th>Prod'n.</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

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

Handles

What are we trying to find?

A substring \( \alpha \) of the tree's upper frontier that

matches some production \( A \rightarrow \alpha \) where reducing \( \alpha \) 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 \( \gamma \) is a production \( A \rightarrow \beta \) and a position in \( \gamma \) where \( \beta \) may be found and replaced by \( A \) to produce the previous right-sentential form in a rightmost derivation of \( \gamma \)

i.e., if \( S \Rightarrow^+ \alpha W \Rightarrow^+ \alpha \beta W \) then \( A \rightarrow \beta \) in the position following \( \alpha \) is a handle of \( \alpha \beta W \)

Because \( \gamma \) is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

Handles

Theorem:

If \( G \) is unambiguous then every right-sentential form has a unique handle.

Proof: (by definition)

1. \( G \) is unambiguous \( \Rightarrow \) rightmost derivation is unique

2. \( \Rightarrow \) a unique production \( A \rightarrow \beta \) applied to take \( \gamma_{i-1} \) to \( \gamma_i \)

3. \( \Rightarrow \) a unique position \( \ell \) at which \( A \rightarrow \beta \) is applied

4. \( \Rightarrow \) a unique handle \( A \rightarrow \beta \)
Example

The left-recursive expression grammar (original form)

<table>
<thead>
<tr>
<th>Prod'n.</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(goal) ::= (exp)</td>
</tr>
<tr>
<td>2</td>
<td>(exp) ::= (exp) + (term)</td>
</tr>
<tr>
<td>3</td>
<td>(exp) ::= (exp) - (term)</td>
</tr>
<tr>
<td>4</td>
<td>(term) ::= (term) * (factor)</td>
</tr>
<tr>
<td>5</td>
<td>(term) ::= (term) / (factor)</td>
</tr>
<tr>
<td>6</td>
<td>(factor) ::= num</td>
</tr>
<tr>
<td>7</td>
<td>(factor) ::= id</td>
</tr>
</tbody>
</table>

Handle-pruning

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

To construct a rightmost derivation

\[ S \rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w \]

we set \( i \) to \( n \) and apply the following simple algorithm

for \( i = n \) downto 1

1. find the handle \( A_i \rightarrow \beta_i \) 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) find the handle
      if we have a handle $A \rightarrow \beta$ on top of the stack, shift an input symbol onto the stack
   b) prune the handle
      if we have a handle $\beta | \beta$ on the stack, reduce
      i) pop $| \beta |$ symbols off the stack
      ii) push $A$ onto the stack

Example: back to $x - 2 \cdot y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$1d$</td>
<td>num + id</td>
</tr>
<tr>
<td>$$$</td>
<td>$$$</td>
<td>shift</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
<tr>
<td>$1g$</td>
<td>$1d$</td>
<td>$1d$</td>
</tr>
</tbody>
</table>

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

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n = w;$$

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 $\gamma$ from left to right, going at most k symbols beyond the right end of the handle of $\gamma_i$.

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

$\text{LL}(k)$: recognize use of a production $A \rightarrow \beta$ seeing first $k$ symbols of $\beta$

$\text{LR}(k)$: recognize occurrence of $\beta$ (the handle) having seen all of what is derived from $\beta$ 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.

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

Example of a token specification:

```
TOKEN :

  < INTEGER_LITERAL : ["1"-"9"] (["0"-"9"]*) | "0" 
```

Example of a production:

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

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 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`.

```java
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.

```java
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.

```java
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();
1.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.

<table>
<thead>
<tr>
<th></th>
<th>Frequent type casts</th>
<th>Frequent recompilation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instanceof and type casts</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Dedicated methods</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The Visitor pattern</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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 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

```
ftb fortran.js // generates jtb.out.js
javacc jtb.out.js // 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)

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 v) {
        v.visit(this);
    }
}
```

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

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)

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:

```java
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.

---

**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 `f` 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:

<table>
<thead>
<tr>
<th>Abstract syntax tree (attribute grammars)</th>
<th>specify non-local computations automatic evaluators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol tables</td>
<td>central store for facts express checking code</td>
</tr>
<tr>
<td>Language design</td>
<td>simplify language avoid problems</td>
</tr>
</tbody>
</table>

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

Separate table for structure layouts (types) (field 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. $\text{array}(\ell, T)$ denotes array of elements type $T$, index type $\ell$
      e.g., $\text{array}(1 \ldots 10, \text{integer})$
   b. $T_1 \times T_2$ denotes Cartesian product of type expressions $T_1$ and $T_2$
   c. records: fields have names
      e.g., $\text{record}(\langle \text{a \times \text{integer}}, \langle \text{b \times real} \rangle)\$
   d. $\text{pointer}(T)$ denotes the type “pointer to object of type $T$”
   e. $D \rightarrow K$ denotes type of function mapping domain $D$ to range $K$
      e.g., $\text{integer \times integer \rightarrow integer}$

Type descriptors

Type descriptors are compile-time structures representing type expressions

E.g., $\text{char} \times \text{char} \rightarrow \text{pointer(integer)}$

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)

- $s = t$ iff. $s$ and $t$ are the same basic types
- $\text{array}(s_1, s_2) \equiv \text{array}(t_1, t_2)$ 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 $s_2 \equiv t_2$
- $\text{pointer}(s) \equiv \text{pointer}(t)$ 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:

```pascal
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/Tiger are somewhat confusing: they treat distinct type definitions as distinct types, so

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

Type compatibility: recursive types

Consider:

```pascal
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:

```
cell = record

  info integer next pointer

  cell
```

Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:

```
cell = record

  info integer next pointer
```

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)

```
next last p q r

link = pointer pointer pointer

Cell
```

Type expressions are equivalent if they are represented by the same node in the graph
Chapter 7: Translation and Simplification

Tiger IR trees: Expressions

<table>
<thead>
<tr>
<th>Expression Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONST</strong></td>
<td>Integer constant i</td>
</tr>
<tr>
<td><strong>NAME</strong></td>
<td>Symbolic constant n [a code label]</td>
</tr>
<tr>
<td><strong>TEMP</strong></td>
<td>Temporary t [one of any number of &quot;registers&quot;]</td>
</tr>
<tr>
<td><strong>BINOP</strong></td>
<td>Application of binary operator ( e \times ): PLUS, MINUS, MUL, DIV, AND, OR, XOR, LSHIFT, RSHIFT, ARSHIFT [bitwise logical, logical shifts, arithmetic right-shift to integer operands ( e_1 ) (evaluated first) and ( e_2 ) (evaluated second)]</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>Contents of a word of memory starting at address e</td>
</tr>
<tr>
<td><strong>CALL</strong></td>
<td>Procedure call; expression f is evaluated before arguments ( e_1, \ldots, e_n )</td>
</tr>
<tr>
<td><strong>SEQ</strong></td>
<td>Expression sequence; evaluate f for side-effects, then ( e ) for result</td>
</tr>
</tbody>
</table>

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 Tiger

Simple variables: fetch with a MEM:

\[
\text{MEM BINOP Ex(MEM(+(TEMP fp, CONST k)))}
\]

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

Tiger array variables: Tiger arrays are pointers to array base, so fetch with a MEM like any other variable:

\[
\text{Ex(MEM(+(TEMP fp, CONST k)))}
\]

Thus, for \( e[i] \):

\[
\text{Ex(MEM(+(e.unEx, x(i.unEx, CONST w))))}
\]

where \( w \) is the word size – all values are word-sized (scalar) in Tiger

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

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

\[ \text{while } c \text{ do } s : \]

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

\[ \text{e.g.,} \]

\[ \text{test:} \]

\[ \text{if not}(c) \text{ jump done} \]

\[ \text{jump test} \]

\[ \text{done:} \]

\[ \text{Nx( SEQ(SEQ(SEQ(LABEL test, \text{unCx(body, done)}), SEQ(SEQ(LABEL body, \text{unNx}), JUMP(NAME test))), LABEL done)))} \]

repeat \( e_1 \) until \( e_2 \) ← evaluate/compare/branch at bottom of loop
for loops

for \( \mathbf{i} := e_1 \text{ to } e_2 \) 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

body: \( s \)

1. \( t_1 \leftarrow e_1 \)
2. \( t_2 \leftarrow e_2 \)
   if \( t_1 > t_2 \) jump done

body: \( s \)

1. \( t_1 \leftarrow t_1 + 1 \)
   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

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

\[ \text{Ex(CALL(NAME label f, [sl f, e_1, \ldots, e_n]))} \]

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 \( a \text{ op } b \) as:

\[ \text{RelCx(op, a.unEx, b.unEx)} \]

When used as a conditional \( \text{unCx}(i, f) \) yields:

\[ \text{CJUMP(op, a.unEx, b.unEx, i, f)} \]

where \( i \) and \( f \) are labels.

When used as a value \( \text{unEx} \) yields:

\[ \text{ESEQ(SEQ(MOVE(TMP r, CONST 1)), SEQ(unCx(t, f)), SEQ(LABEL l), SEQ(MOVE(TMP r, CONST 0), LABEL m)), TEMP r)} \]

Conditionals

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

- e.g., \( x < 5 \text{ & } a > b \) turns into \( x < 5 \text{ then } a > b \text{ else } 0 \)

Translate if \( e_1 \text{ then } e_2 \text{ else } e_3 \) into: \( \text{IfThenElseExp} \( e_1, e_2, e_3 \) \)

When used as a value \( \text{unEx} \) yields:

\[ \text{ESEQ(SEQ(SEQ(MOVE(TMP r, CONST 1)), SEQ(MOVE(TMP f, e_2.unEx)), JUMP join)), SEQ(LABEL l), SEQ(MOVE(TMP r, e_3.unEx)), JUMP join)), LABEL join), TEMP r} \]

As a conditional \( \text{unCx}(i, f) \) yields:

\[ \text{SEQ(e_1.unCx(tt, ff)), SEQ(SEQ(LABEL tt, e_2.unCx(i, f))), SEQ(LABEL ff, e_3.unCx(i, f)))} \]
Conditionals: Example

Applying unCx(r, 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, r, 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

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

translates to:
MEM(+(TEMP fp, +(CONST k-2w, x(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[1,,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, \ldots, N\} of \( T \)
= array \{1, \ldots, N\} of array \{1, \ldots, N\} of \( T \)
no. of elt's in dimension \( j \):
\[
D_j = U_j - L_j + 1
\]
position of \( A[i_1, \ldots, i_n] \):
\[
\begin{align*}
&\{(a - L_0) + (i_{n-1} - L_{n-1})D_{n-1} + (i_{n-2} - L_{n-2})D_{n-2} + \cdots + (i_1 - L_1)D_2 \cdot D_1\} \\
&\text{address(A)} + ((\text{variable part} - \text{constant part}) \times \text{element size})
\end{align*}
\]

case statements

\textbf{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)
  \( O(\text{cases}) \)
- binary search of an ordered jump table (sparse case set)
  \( O(\log_{\text{cases}}) \)
- hash table (dense case set)
  \( O(1) \)

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

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

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESEQ(( a ), ESEQ(( b ), ( c )) )</td>
<td>= ESEQ(SEQ(( a ), SEQ(( b ), ( c )) )</td>
</tr>
<tr>
<td>BINOPT(op, ESEQ(( a ), ( b )) )</td>
<td>= SEQ(SEQ(( a ), BINOPT(( op ), ( b )) )</td>
</tr>
<tr>
<td>MEM(ESEQ(( a ), ( e )) )</td>
<td>= ESEQ(SEQ(( a ), MEM(( e )) )</td>
</tr>
<tr>
<td>JUMP(ESEQ(( a ), ( e )) )</td>
<td>= SEQ(SEQ(( a ), JUMP(( e )) )</td>
</tr>
<tr>
<td>CJUMP(op, ESEQ(( a ), ( b )) )</td>
<td>= SEQ(SEQ(( a ), CJUMP(( op ), ( b )) )</td>
</tr>
<tr>
<td>JUMP(op, ( a ), ESEQ(( b ), ( c )) )</td>
<td>= SEQ(SEQ(( a ), JUMP(( op ), ( b ), ( c )) )</td>
</tr>
<tr>
<td>BINOP(op, ( a ), ESEQ(( b ), ( c )) )</td>
<td>= SEQ(SEQ(( a ), BINOP(( op ), ( b ), ( c )) )</td>
</tr>
<tr>
<td>CJUMP(op, ( a ), ESEQ(( b ), ( c )) )</td>
<td>= SEQ(SEQ(( a ), CJUMP(( op ), ( b ), ( c )) )</td>
</tr>
<tr>
<td>MOVE(SEQ(( a ), ( b )) )</td>
<td>= SEQ(SEQ(( a ), CALL(( f ), ( a )) , ( b )) )</td>
</tr>
<tr>
<td>CALL(( f ), ( a ))</td>
<td>= ESEQ(MOVE(TEMP(( f )), ( a )) , ( b ))</td>
</tr>
</tbody>
</table>
Chapter 8: Liveness Analysis and 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
  \[\text{NP-complete for } k \geq 1 \text{ registers}\]

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, \( \text{succ}[n] \)
In-edges to node \( n \) come from predecessor nodes, \( \text{pred}[n] \)

Example:

\[
L_1: \quad a \leftarrow 0 \\
b \leftarrow a + 1 \\
c \leftarrow c + b \\
a \leftarrow b \times 3 \\
\text{if } a < N \text{ goto } L_1 \\
\text{return } c
\]
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 define a variable, \( v \):
  - \( \text{def}_v \) — set of graph nodes that define \( v \)
  - \( \text{use}_v \) — set of variables defined by \( v \)
- occurrences of \( v \) in expressions use it:
  - \( \text{use}_v \) — set of nodes that use \( v \)
  - \( \text{use}_v \) — set of variables used in \( v \)

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 \( \text{def}_v \)

\( v \) 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

\( v \in \text{use}_n \) \( \Rightarrow \) \( v \) live-in at \( n \)

\( v \) live-in at \( n \) \( \Rightarrow \) \( v \) live-out at all \( m \in \text{pred}_n \)

\( v \) live-out at \( n \), \( v \notin \text{def}_n \) \( \Rightarrow \) \( v \) live-in at \( n \)

Iterative solution for liveness

```c
foreach n [ \( \text{in}[n] \leftarrow \emptyset; \text{out}[n] \leftarrow \emptyset \) ]
repeat
  foreach n
    \( \text{in}[n] \leftarrow \text{in}[n]; \)
    \( \text{out}[n] \leftarrow \text{out}[n]; \)
    \( \text{in}[n] \leftarrow \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]); \)
    \( \text{out}[n] \leftarrow \bigcup_{n \in \text{succ}[n]} \text{in}[n]; \)
  until \( \text{in}[n] \leftarrow \text{in}[n] \setminus \text{out}[n]; \forall n \)
```

Notes:
- should order computation of inner loop to follow the “flow”
- liveness flows backward along control-flow arcs, from \( \text{out} \) to \( \text{in} \)
- 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

Liveness analysis

Define:
\( \text{in}[n] \): variables live-in at \( n \)  
\( \text{out}[n] \): variables live-out at \( n \)

Then:
\[
\begin{align*}
\text{out}[n] & = \bigcup_{n \in \text{succ}[n]} \text{in}[n] \\
\text{succ}[n] & = \emptyset \Rightarrow \text{out}[n] = \emptyset
\end{align*}
\]

Note:
\( \text{in}[n] \supseteq \text{use}[n] \)

\( \text{in}[n] \supseteq \text{out}[n] \setminus \text{def}[n] \)

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

Now, \( v \in \text{in}[n] \iff v \in \text{use}[n] \) or \( v \in \text{out}[n] \setminus \text{def}[n] \)

Thus, \( \text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \)

Iterative solution for liveness

Complexity: for input program of size \( N \)

- \( \leq N \) nodes in CFG  
  \( \Rightarrow \leq N \) variables  
  \( \Rightarrow N \) elements per \( \text{in/out} \)  
  \( \Rightarrow O(N) \) time per set-union
- for loop performs constant number of set operations per node  
  \( \Rightarrow O(N^2) \) time for for loop
- each iteration of repeat loop can only add to each set  
  each set can contain at most every variable  
  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 \( O(N^3) \)
- ordering can cut repeat loop down to 2-3 iterations  
  \( \Rightarrow O(N) \) or \( O(N^2) \) in practice
Iterative solution for liveness

Least fixed 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 $n$  
  \[ \Rightarrow v \in \text{out}(n) \]
- 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
  \[ \Rightarrow \text{NP-complete for } k \geq 1 \text{ registers} \]

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
   (d) leads to recursive coloring algorithm

3. Spill: suppose $m$ 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 significant 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 simplification 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 simplified
   - repeat simplify and coalesce until only significant-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 finished, otherwise
   (a) rewrite code to fetch actual spills before each use and store after each definition
   (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

```plaintext
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/residual)
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:

  ![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:

<table>
<thead>
<tr>
<th>Node</th>
<th>uses + defs</th>
<th>uses + defs</th>
<th>degree</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2 +10x 0</td>
<td>4</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1 +10x 1</td>
<td>4</td>
<td>2.75</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2 +10x 0</td>
<td>4</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2 +10x 2</td>
<td>4</td>
<td>5.50</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1 +10x 3</td>
<td>3</td>
<td>10.30</td>
<td></td>
</tr>
</tbody>
</table>

- Node c has lowest priority so spill it

Example (cont.)

- Interference graph with c removed:

  ![Interference Graph]

- 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 \( r_2 \) (or coalesce \( a \) and \( r_1 \)):

- Coalescing \( a \) and \( r_1 \) (could also coalesce \( d \) with \( r_1 \)):

Example (cont.)

- Cannot coalesce \( r_1a \) 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
  - \( d \equiv r_3 \)
  - \( a, b, e \) have colors by coalescing
  - \( c \) must spill since no color can be found for it
- Introduce new temporaries \( c_1 \) and \( c_2 \) for each use/def, add loads before each use and stores after each def

---

Example (cont.)

```
enter:
c1 := r3
M[x_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[x_loc]
r3 := c2
return [r1, r3 live out ]
```

Example (cont.)

- New interference graph:

- Coalesce \( c_1 \) with \( r_3 \), then \( c_2 \) with \( r_3 \):

- As before, coalesce \( a \) with \( e \), then \( b \) with \( r_2 \):
As before, coalesce \texttt{ae} with \texttt{r1} and simplify \texttt{d}:

\[
\begin{array}{c}
\texttt{a} \\
\texttt{b} \\
\texttt{c}
\end{array}
\]

Pop \texttt{d} from stack: select \texttt{r3}. All other nodes were coalesced or precolored. So, the coloring is:

- \texttt{a} \equiv \texttt{r1}
- \texttt{b} \equiv \texttt{r2}
- \texttt{c} \equiv \texttt{r3}
- \texttt{d} \equiv \texttt{r3}
- \texttt{e} \equiv \texttt{r1}

Example (cont.)

- Rewrite the program with this assignment:

```plaintext
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):

```plaintext
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.

Code to make the linkage is generated at compile time

Copyright © 2000 by Antony L. Hoising. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hoising@cs.purdue.edu.

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

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

---

**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

Two important problems arise

* How do we map a name into a \((level,offset)\) pair?
  Use a block-structured symbol table
    * 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
    * displays

Real globals

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

Lexical nesting

* view variables as \((level,offset)\) pairs

Chain of non-local access links

* more expensive to find

Access to non-local data

To find the value specified by \((i,o)\)

* need current procedure level, \(k\)
* \(k - i = \) local value
* \(k > i \Rightarrow \) find \(i\)’s activation record
* \(k < i \) cannot occur

Maintaining access 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 \( \text{display}[l] \)
- add offset to pointer from slot \( \text{display}/[l][o] \)

“Setting up the basic frame” now includes display manipulation

Calls: Saving and restoring registers

<table>
<thead>
<tr>
<th>callee saves</th>
<th>caller’s registers</th>
<th>callee’s registers</th>
<th>all registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

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 actuals
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

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

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

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-$sp
3. Get return address
   lw $31, framesize+frameoffset($sp)
4. Clean up stack
   adds $sp, framesize
5. Return
   j $31