CS 132 Compiler Construction

Chapter 1: Introduction

Things to do

- Brush up on Java
- CCLE
- Piazza

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

Compiler construction is a microcosm of computer science

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

Inside a compiler, all these things come together

Isn't it a solved problem?

Machines are constantly changing

Changes in architecture $\Rightarrow$ changes in compilers

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

Changes in compilers should prompt changes in architecture

- New languages and features

Intrinsic Merit

Compiler construction is challenging and fun

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

Compilers have an impact on how computers are used

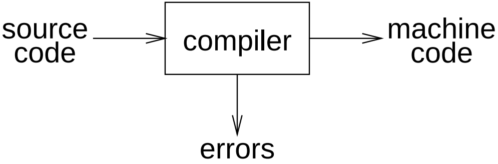
Compiler construction poses interesting problems

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

Abstract view

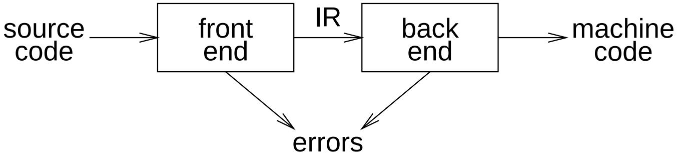


Implications:

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

Big step up from assembler to higher level notations

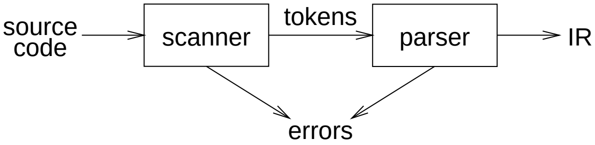
Two pass compiler



Implications:

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

Front end

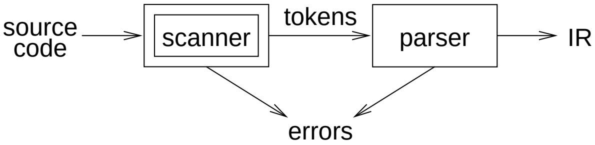


Responsibilities:

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

Much of front end construction can be automated

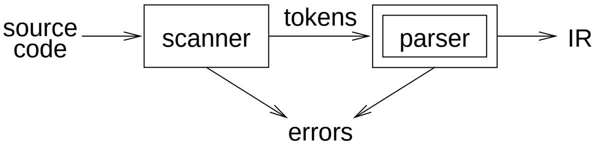
Front end



Scanner:

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

Front end

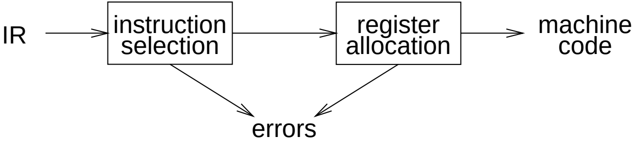


Parser:

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

Parser generators mechanize much of the work

Back end

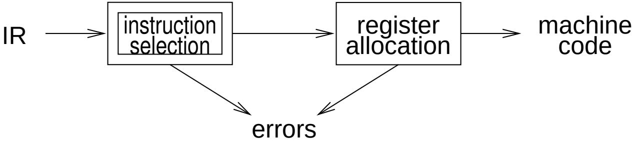


Responsibilities

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

Automation has been less successful here

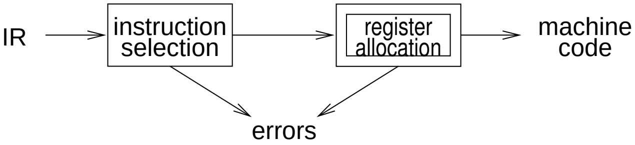
Back end



Instruction selection:

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

Back end

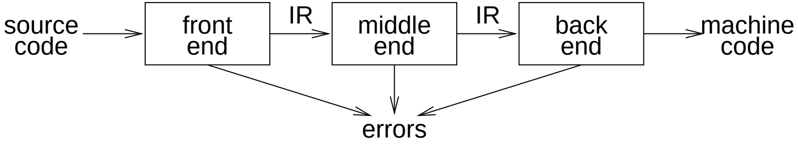


Register Allocation:

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

Modern allocators often use an analogy to graph coloring

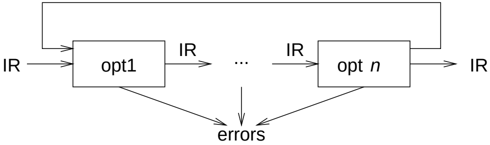
Optimizing compiler



Code Improvement

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

Optimizer (middle end)

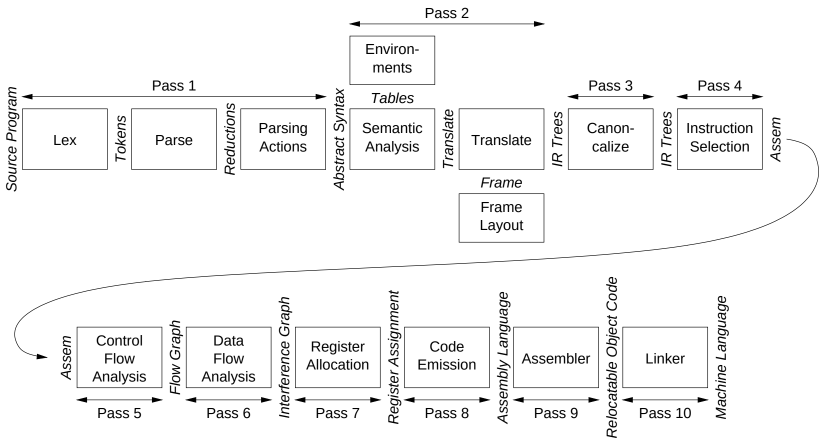


Modern optimizers are usually built as a set of passes

Typical passes

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

Compiler example

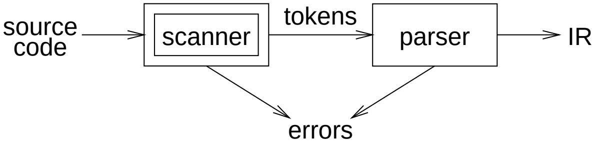


Compiler phases

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

Chapter 2: Lexical Analysis

Scanner



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

  `x = x + y;`

  becomes

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

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

Specifying patterns

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

Some parts are easy:
white space

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

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

comments
opening and closing delimiters: `/* ··· */`

Specifying patterns

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

Other parts are much harder:

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

numbers

- integers: 0 or digit from 1-9 followed by digits from 0-9
- decimals: integer `'.'` digits from 0-9
- reals: (integer or decimal) `'E'` (+ or -) digits from 0-9
- complex: `'('` real `','` real `')'`

We will use regular expressions to specify these patterns

CS 181 is a Requisite of CS 132

- Regular languages and regular expressions
- Algebraic properties of regular expressions
- Nondeterministic finite automata (NFA)
- Deterministic finite automata (DFA)
- The mapping from regular expressions to NFA
- The mapping from NFA to DFA (the subset construction)

Examples

identifier	$\textit{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$
	$\textit{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$
	$\textit{id} \rightarrow \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^*$
numbers	$\textit{integer} \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9)\ \textit{digit}^*)$
	$\textit{decimal} \rightarrow \textit{integer}\ \mathbf{.}\ (\ \textit{digit}\ )^*$
	$\textit{real} \rightarrow (\ \textit{integer} \mid \textit{decimal}\ )\ \mathtt{E}\ (+ \mid -)\ \textit{digit}^*$
	$\textit{complex} \rightarrow \mathtt{'('}\ \textit{real}\ ,\ \textit{real}\ \mathtt{')'}$

Numbers can get more complicated

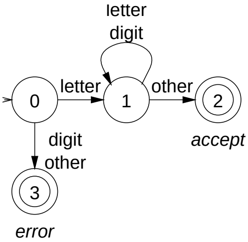
Most programming language tokens can be described with REs
We can use REs to build scanners automatically

Recognizers

From a regular expression we can construct a

deterministic finite automaton (DFA)

Recognizer for *identifier*:



identifier

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

Code for the recognizer

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

Tables for the recognizer

Two tables control the recognizer

`char_class:`

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

`next_state:`

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

To change languages, we can just change tables

Automatic construction

Scanner generators automatically construct code from regular expression-like descriptions

- construct a *dfa*
- use state minimization techniques
- emit code for the scanner
  (table driven or direct code)

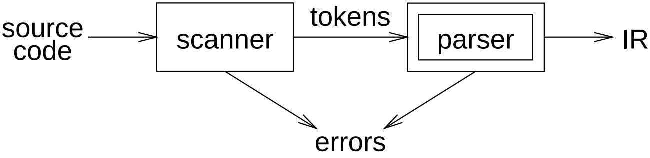
A key issue in automation is an interface to the parser

`lex` is a scanner generator supplied with UNIX

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

Chapter 3: LL Parsing

The role of the parser



Parser

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

Syntax analysis

Context-free syntax is specified with a *context-free grammar*.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

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

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

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

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

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

Notation and terminology

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

If $A \to \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \to \gamma$
Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps
If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G
$L(G) = \{w \in V_t^* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a *sentence* of G
Note, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Syntax analysis

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

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

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

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

Scanning vs. parsing

Where do we draw the line?

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

Regular expressions are used to classify:

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

Context-free grammars are used to count:

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

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

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG:

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

We have derived the sentence $\texttt{x} + 2 * \texttt{y}$.
We denote this $\langle\text{goal}\rangle \Rightarrow^* \texttt{id} + \texttt{num} * \texttt{id}$.
Such a sequence of rewrites is a *derivation* or a *parse*.
The process of discovering a derivation is called *parsing*.

Derivations

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

This choice can lead to different derivations.

Two are of particular interest:

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

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

The previous example was a leftmost derivation.

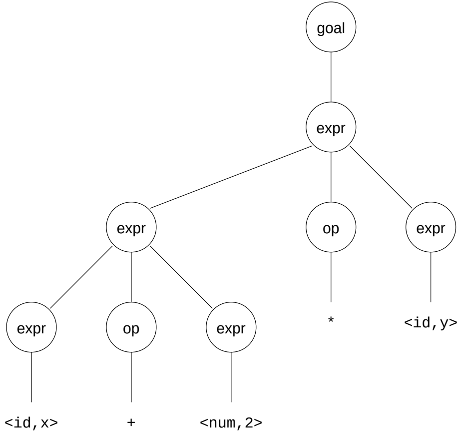
Rightmost derivation

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

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

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

Precedence



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

Precedence

These two derivations point out a problem with the grammar.
It has no notion of precedence, or implied order of evaluation.
To add precedence takes additional machinery:

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

This grammar enforces a precedence on the derivation:

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

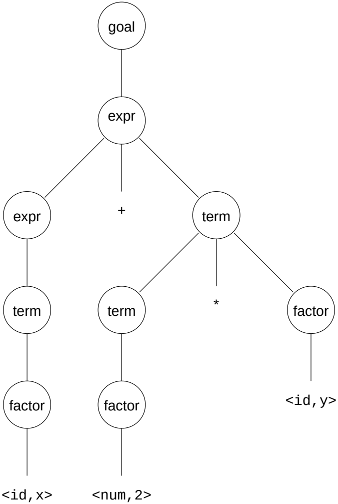
Precedence

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

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

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

Precedence



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

Ambiguity

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

Example:

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

Consider deriving the sentential form:

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

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

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

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

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

This is most likely the language designer's intent.

Ambiguity

Ambiguity is often due to confusion in the context-free specification.
Context-sensitive confusions can arise from *overloading*.
Example:

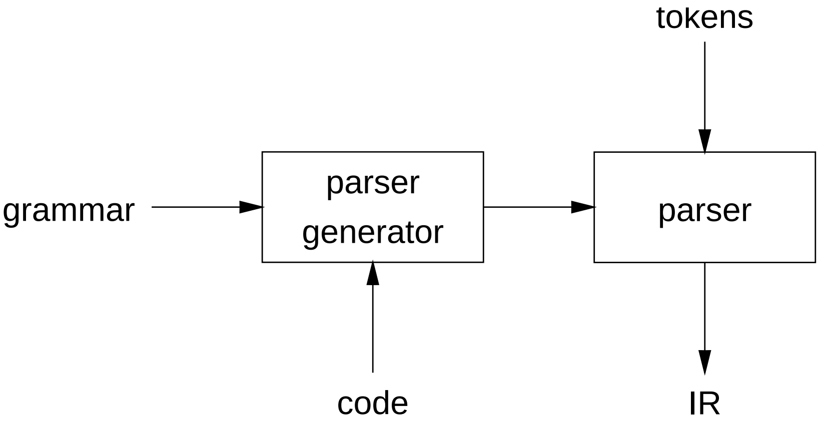
```
a = f(17)
```

In many Algol-like languages, `f` could be a function or subscripted variable.
Disambiguating this statement requires context:

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

Rather than complicate parsing, we will handle this separately.

Parsing: the big picture



Our goal is a flexible parser generator system

Top-down versus bottom-up

Top-down parsers

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

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid 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 α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find the next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1

$\Rightarrow$ *should be guided by input string*

Simple expression grammar

Recall our grammar for simple expressions:

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

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

Example

Prod'n	Sentential form	Input					
–	$\langle$goal$\rangle$	↑x	–	2	$*$	y	
1	$\langle$expr$\rangle$	↑x	–	2	$*$	y	
2	$\langle$expr$\rangle$ + $\langle$term$\rangle$	↑x	–	2	$*$	y	
4	$\langle$term$\rangle$ + $\langle$term$\rangle$	↑x	–	2	$*$	y	
7	$\langle$factor$\rangle$ + $\langle$term$\rangle$	↑x	–	2	$*$	y	
9	id + $\langle$term$\rangle$	↑x	–	2	$*$	y	
–	id + $\langle$term$\rangle$	x	↑–	2	$*$	y	
–	$\langle$expr$\rangle$	↑x	–	2	$*$	y	
3	$\langle$expr$\rangle$ – $\langle$term$\rangle$	↑x	–	2	$*$	y	
4	$\langle$term$\rangle$ – $\langle$term$\rangle$	↑x	–	2	$*$	y	
7	$\langle$factor$\rangle$ – $\langle$term$\rangle$	↑x	–	2	$*$	y	
9	id – $\langle$term$\rangle$	↑x	–	2	$*$	y	
–	id – $\langle$term$\rangle$	x	↑–	2	$*$	y	
–	id – $\langle$term$\rangle$	x	–	↑2	$*$	y	
7	id – $\langle$factor$\rangle$	x	–	↑2	$*$	y	
8	id – num	x	–	↑2	$*$	y	
–	id – num	x	–	2	↑$*$	y	
–	id – $\langle$term$\rangle$	x	–	↑2	$*$	y	
5	id – $\langle$term$\rangle$ $*$ $\langle$factor$\rangle$	x	–	↑2	$*$	y	
7	id – $\langle$factor$\rangle$ $*$ $\langle$factor$\rangle$	x	–	↑2	$*$	y	
8	id – num $*$ $\langle$factor$\rangle$	x	–	↑2	$*$	y	
–	id – num $*$ $\langle$factor$\rangle$	x	–	2	↑$*$	y	
–	id – num $*$ $\langle$factor$\rangle$	x	–	2	$*$	↑y	
9	id – num $*$ id	x	–	2	$*$	↑y	
–	id – num $*$ id	x	–	2	$*$	y	↑

Example

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

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

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

Left-recursion

Top-down parsers cannot handle left-recursion in a grammar
Formally, a grammar is *left-recursive* if

$$\exists A \in V_n \text{ such that } A \Rightarrow^{+} A\alpha \text{ for some string } \alpha$$

Our simple expression grammar is left-recursive

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

Consider the grammar fragment:

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

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

We can rewrite this as:

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

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

This fragment contains no left-recursion

Example

Our expression grammar contains two cases of left-recursion

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

Applying the transformation gives

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

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

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

It is

- right-recursive
- free of ε productions

Unfortunately, it generates different associativity
Same syntax, different meaning

Example

Our long-suffering expression grammar:

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

Recall, we factored out left-recursion

How much lookahead is needed?

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

Do we need arbitrary lookahead to parse CFGs?

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

  Aho, Hopcroft, and Ullman, Problem 2.34

  Parsing, Translation and Compiling, Chapter 4

Fortunately

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

Among the interesting subclasses are:

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

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

Predictive parsing

Basic idea:

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

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

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

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

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

The example grammar has this property!

Left factoring

What if a grammar does not have this property?
Sometimes, we can transform a grammar to have this property.

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

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

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

where A' is a new non-terminal.

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

Example

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

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

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

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

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals that must be left factored:

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

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

Applying the transformation gives us:

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

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

Example

Substituting back into the grammar yields

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

Now, selection requires only a single token lookahead.

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

Example

	Sentential form	Input
–	$\langle$goal$\rangle$	$\uparrow$x – 2 * y
1	$\langle$expr$\rangle$	$\uparrow$x – 2 * y
2	$\langle$term$\rangle\langle$expr$'\rangle$	$\uparrow$x – 2 * y
6	$\langle$factor$\rangle\langle$term$'\rangle\langle$expr$'\rangle$	$\uparrow$x – 2 * y
11	id$\langle$term$'\rangle\langle$expr$'\rangle$	$\uparrow$x – 2 * y
–	id$\langle$term$'\rangle\langle$expr$'\rangle$	x $\uparrow$- 2 * y
9	idε $\langle$expr$'\rangle$	x $\uparrow$- 2
4	id– $\langle$expr$\rangle$	x $\uparrow$- 2 * y
–	id– $\langle$expr$\rangle$	x – $\uparrow$2 * y
2	id– $\langle$term$\rangle\langle$expr$'\rangle$	x – $\uparrow$2 * y
6	id– $\langle$factor$\rangle\langle$term$'\rangle\langle$expr$'\rangle$	x – $\uparrow$2 * y
10	id– num$\langle$term$'\rangle\langle$expr$'\rangle$	x – $\uparrow$2 * y
–	id– num$\langle$term$'\rangle\langle$expr$'\rangle$	x – 2 $\uparrow$* y
7	id– num* $\langle$term$\rangle\langle$expr$'\rangle$	x – 2 $\uparrow$* y
–	id– num* $\langle$term$\rangle\langle$expr$'\rangle$	x – 2 * $\uparrow$y
6	id– num* $\langle$factor$\rangle\langle$term$'\rangle\langle$expr$'\rangle$	x – 2 * $\uparrow$y
11	id– num* id$\langle$term$'\rangle\langle$expr$'\rangle$	x – 2 * $\uparrow$y
–	id– num* id$\langle$term$'\rangle\langle$expr$'\rangle$	x – 2 * y$\uparrow$
9	id– num* id$\langle$expr$'\rangle$	x – 2 * y$\uparrow$
5	id– num* id	x – 2 * y$\uparrow$

The next symbol determined each choice correctly.

Back to left-recursion elimination

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

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

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

with

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

where N and A' are new productions.

Repeat until there are no left-recursive productions.

Generality

Question:

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

Answer:

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

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

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

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

Recursive descent parsing

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

```
Token token;
void eat(char a) {
   if (token == a){ token = next_token(); }
                  { error(); }
}
void goal() { token = next_token(); expr(); eat(EOF); }
void expr() { term(); expr_prime(); }
void expr_prime() {
   if (token == PLUS)
      { eat(PLUS); expr(); }
   else if (token == MINUS)
      { eat(MINUS); expr(); }
   else { }
}
```

Recursive descent parsing

```
void term() { factor(); term_prime(); }
void term_prime() {
   if (token = MULT)
      { eat(MULT); term(); }
   else if (token = DIV)
      { eat(DIV); term(); }
   else { }
}
void factor() {
   if (token = NUM)
      { eat(NUM); }
   else if (token = ID)
      { eat(ID); }
   else error();
}
```

Nullable

For a string α of grammar symbols, define $\text{NULLABLE}(\alpha)$ as

> α *can go to* ε.

$\text{NULLABLE}(\alpha)$ if and only if $(\alpha \Rightarrow^* \varepsilon)$

How to compute $\text{NULLABLE}(U)$, for $U \in V_t \cup V_n$.

1. For each U, let $\text{NULLABLE}(U)$ be a Boolean variable.
2. Derive the following constraints:
   2.1 If $a \in V_t$,
      - $\text{NULLABLE}(a) = \text{false}$

   2.2 If $A \to Y_1 \cdots Y_k$ is a production:
      - $[\,\text{NULLABLE}(Y_1) \wedge \cdots \wedge \text{NULLABLE}(Y_k)\,] \Longrightarrow \text{NULLABLE}(A)$
3. Solve the constraints.

$$\text{NULLABLE}(X_1 \cdots X_k) \;=\; \text{NULLABLE}(X_1) \wedge \cdots \wedge \text{NULLABLE}(X_k)$$

FIRST

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

> *the set of terminal symbols that begin strings derived from α.*

$$\textsc{first}(\alpha) \;=\; \{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$$

How to compute $\textsc{first}(U)$, for $U \in V_t \cup V_n$.

1. For each U, let $\textsc{first}(U)$ be a set variable.
2. Derive the following constraints:
   2.1 If $a \in V_t$,
   - $\textsc{first}(a) = \{\, a \,\}$

   2.2 If $A \rightarrow Y_1 Y_2 \cdots Y_k$ is a production:
   - $\textsc{first}(Y_1) \subseteq \textsc{first}(A)$
   - $\forall i : 1 < i \leq k$, if $\textsc{Nullable}(Y_1 \cdots Y_{i-1})$, then $\textsc{first}(Y_i) \subseteq \textsc{first}(A)$
3. Solve the constraints. Go for the $\subseteq$-least solution.

$$\textsc{first}(X_1 \cdots X_k) \;=\; \bigcup_{i:1\leq i\leq k \wedge \textsc{nullable}(X_1 \cdots X_{i-1})} \textsc{first}(X_i)$$

FOLLOW

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

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

$$\text{FOLLOW}(B) \;=\; \{a \in V_t \mid G \Rightarrow^* \alpha B \beta \wedge a \in \text{FIRST}(\beta\,\$)\}$$

How to compute $\text{FOLLOW}(B)$.

1. For each non-terminal B, let $\text{FOLLOW}(B)$ be a set variable.
2. Derive the following constraints:
   2.1 If G is the start symbol and \$ is the end-of-file marker, then
      - $\{\,\$\,\} \subseteq \text{FOLLOW}(G)$

   2.2 If $A \rightarrow \alpha B \beta$ is a production:
      - $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(B)$
      - if $\text{NULLABLE}(\beta)$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$
3. Solve the constraints. Go for the $\subseteq$-least solution.

LL(1) grammars

Intuition: A grammar G is LL(1) iff
for all non-terminals A, each distinct pair of productions

$A \to \beta$
$A \to \gamma$

satisfy the condition $\textsc{first}(\beta) \bigcap \textsc{first}(\gamma) = \emptyset$.

Question: What if $\textsc{nullable}(A)$?

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

1. $\textsc{first}(\alpha_1), \textsc{first}(\alpha_2), \ldots, \textsc{first}(\alpha_n)$ are pairwise disjoint, and
2. If $\textsc{nullable}(\alpha_i)$, then for all j, such that $1 \leq j \leq n \wedge j \neq i$:
   $\textsc{first}(\alpha_j) \bigcap \textsc{follow}(A) = \emptyset$.

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

LL(1) grammars

Provable facts about LL(1) grammars:

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

Example

$S \rightarrow aS \mid a$

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

$S \rightarrow aS'$

$S' \rightarrow aS' \mid \varepsilon$

accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G
Output: Parsing table M
Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   1.1 $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
   1.2 If $\varepsilon \in \text{FIRST}(\alpha)$:
      1.2.1 $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
      1.2.2 If $\$ \in \text{FOLLOW}(A)$ then add $A \rightarrow \alpha$ to $M[A, \$]$
2. Set each undefined entry of M to `error`

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

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

Example

Our long-suffering expression grammar:

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

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

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

A grammar that is not LL(1)

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

Left-factored:

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

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

$$\langle \mathit{stmt}' \rangle \ ::= \ \mathtt{else} \ \langle \mathit{stmt} \rangle \quad \textit{and} \quad \langle \mathit{stmt}' \rangle \ ::= \ \varepsilon$$

$\Rightarrow$ grammar is not LL(1)!
The fix:

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

Chapter 4: JavaCC and JTB

The Java Compiler Compiler

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

The JavaCC input format

One file:

- header
- token specifications for lexical analysis
- grammar

The JavaCC input format

Example of a token specification:

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

Example of a production:

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

Generating a parser with JavaCC

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

The Visitor Pattern

For **object-oriented programming**,

the Visitor pattern **enables**

the definition of a **new operation**

on an **object structure**

without changing the classes

of the objects.

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

Sneak Preview

When using the **Visitor** pattern,

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

First Approach: Instanceof and Type Casts

The running Java example: summing an integer list.

```
interface List {}

class Nil implements List {}

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

First Approach: Instanceof and Type Casts

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

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

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

Second Approach: Dedicated Methods

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

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

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

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

Second Approach: Dedicated Methods

```
class Nil implements List {
  public int sum() {
    return 0;
  }
}
class Cons implements List {
  int head;
  List tail;
  public int sum() {
    return head + tail.sum();
  }
}
```

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

Disadvantage: For each new operation on `List`-objects, write new dedicated methods and recompile all classes.

Third Approach: The Visitor Pattern

The Idea:

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

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

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

Third Approach: The Visitor Pattern

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

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

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

Third Approach: The Visitor Pattern

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

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

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

Comparison

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

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

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

Tools that use the Visitor pattern:

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

Visitors: Summary

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

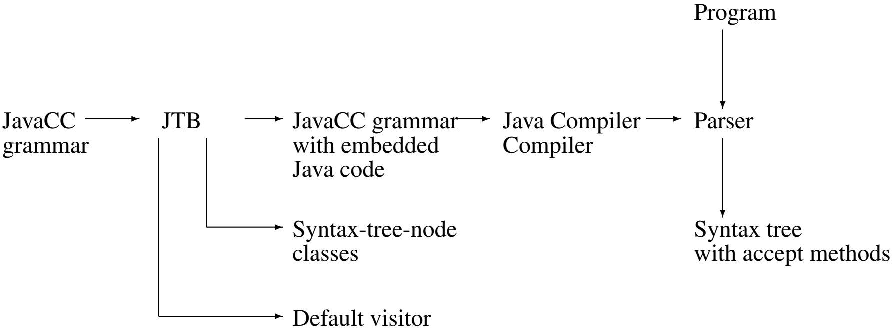
The Java Tree Builder

The Java Tree Builder (JTB) has been developed here at Purdue in my group.
JTB is a frontend for The Java Compiler Compiler.
JTB supports the building of syntax trees which can be traversed using visitors.
JTB transforms a bare JavaCC grammar into three components:

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

The Java Tree Builder

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.
The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Using JTB

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

Example (simplified)

For example, consider the Java 1.1 production

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

JTB produces:

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

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

Example (simplified)

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

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

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

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

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

Example (simplified)

The default visitor looks like this:

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

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

Example (simplified)

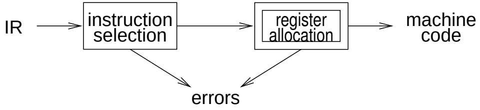
Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

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

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visit` in `VprintAssignRHS` is executed.
Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.
JTB is bootstrapped.

Chapter 5: Liveness Analysis

Register allocation



Register allocation:

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

Liveness analysis

Problem:

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

Approach:

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

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

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

Control flow analysis

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

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

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

$$
\begin{array}{ll}
 & a \leftarrow 0 \\
L_1: & b \leftarrow a + 1 \\
 & c \leftarrow c + b \\
 & a \leftarrow b \times 2 \\
 & \text{if } a < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
$$

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:
  - $def(v)$ = set of graph nodes that define v
  - $def[n]$ = set of variables defined by n
- occurrences of v in expressions *use* it:
  - $use(v)$ = set of nodes that use v
  - $use[n]$ = set of variables used in n

Liveness: v is *live* on edge e if there is a directed path from e to a *use* of v that does not pass through any $def(v)$
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
$v \in use[n] \Rightarrow v$ live-in at n
v live-in at $n \Rightarrow v$ live-out at all $m \in pred[n]$
v live-out at $n, v \notin def[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:
$in[n]$: variables live-in at n
$in[n]$: variables live-out at n
Then:

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

$$succ[n] = \phi \Rightarrow out[n] = \phi$$

Note:

$$in[n] \supseteq use[n]$$

$$in[n] \supseteq out[n] - def[n]$$

$use[n]$ and $def[n]$ are constant (independent of control flow)
Now, $v \in in[n]$ iff. $v \in use[n]$ or $v \in out[n] - def[n]$
Thus, $in[n] = use[n] \cup (out[n] - def[n])$

Iterative solution for liveness

foreach n $\{ in[n] \leftarrow \phi; \;\; out[n] \leftarrow \phi \}$
repeat
    foreach n

$$in'[n] \leftarrow in[n];$$
$$out'[n] \leftarrow out[n];$$
$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$
$$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$$

until $in'[n] = in[n] \wedge out'[n] = out[n], \forall n$

Notes:

- should order computation of inner loop to follow the "flow"
- liveness flows *backward* along control-flow arcs, from *out* to *in*
- nodes can just as easily be basic blocks to reduce CFG size

The Time Complexity of Liveness Analysis

$O(n)$	statements
$O(n)$	variables
$O(n^2)$	iterations
$O(n)$	set unions per iteration
$O(n)$	time to do set union
$O(n^4)$	total

Chapter 6: Activation Records

The procedure abstraction

Separate compilation:

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

The linkage convention:

- a social contract
- machine dependent
- division of responsibility

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

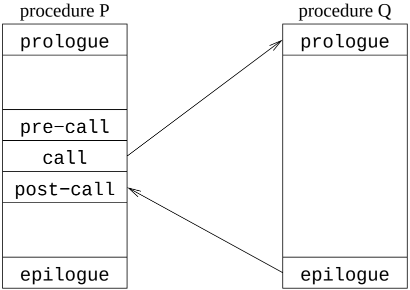
Linkages execute at *run time*

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

The procedure abstraction

The essentials:

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



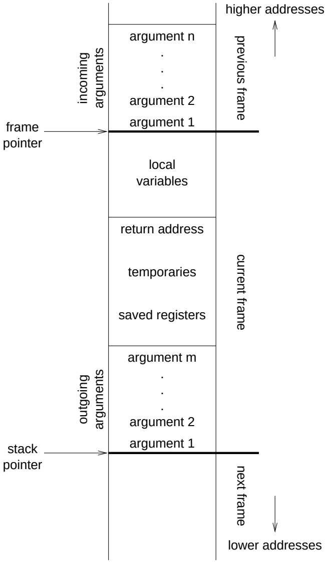
Each system has a *standard linkage*

Procedure linkages

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

Assumptions:

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



Procedure linkages

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

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

Run-time storage organization

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

Code space

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

Data space

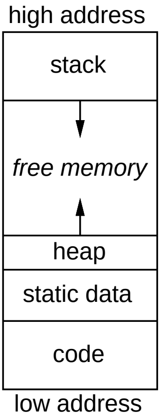
- fixed-sized data may be statically allocated
- variable-sized data must be dynamically allocated
- some data is dynamically allocated in code

Control stack

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

Run-time storage organization

Typical memory layout



The classical scheme

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

Calls: Saving and restoring registers

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

1. Call includes bitmap of caller's registers to be saved/restored (best with save/restore instructions to interpret bitmap directly)
2. Caller saves and restores its own registers. Unstructured returns (e.g., non-local gotos, exceptions) create some problems, since code to restore must be located and executed
3. Backpatch code to save registers used in callee on entry, restore on exit; e.g., VAX places bitmap in callee's stack frame for use on call/return/etc. 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

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

Registers:

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

MIPS procedure call convention

Philosophy:

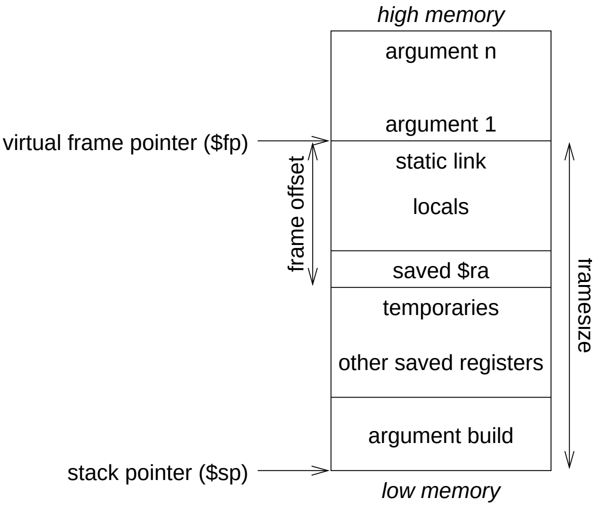
> *Use full, general calling sequence only when necessary; omit portions of it where possible (e.g., avoid using fp register whenever possible)*

Classify routines as:

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

MIPS procedure call convention

The stack frame



MIPS procedure call convention

Pre-call:

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

MIPS procedure call convention

Prologue:

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

      ```
      subu $sp,framesize
      ```
   1.2 Save registers (ra, etc.)
      e.g.,
      ```
      sw $31,framesize+frameoffset($sp)
      sw $17,framesize+frameoffset-4($sp)
      sw $16,framesize+frameoffset-8($sp)
      ```
      where `framesize` and `frameoffset` (usually negative) are compile-time constants
2. Emit code for routine

MIPS procedure call convention

Epilogue:

1. Copy return values into result registers (if not already there)
2. Restore saved registers
   ```
   lw reg,framesize+frameoffset-N($sp)
   ```
3. Get return address
   ```
   lw $31,framesize+frameoffset($sp)
   ```
4. Clean up stack
   ```
   addu $sp,framesize
   ```
5. Return
   ```
   j $31
   ```

Chapter 7: LR Parsing

Some definitions

Recall

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

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

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

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

Bottom-up parsing

Goal:

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

The parser repeatedly matches a *right-sentential* form from the language against the tree's upper frontier.
At each match, it applies a *reduction* to build on the frontier:

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

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

$$
\begin{array}{c|lcl}
1 & S & \rightarrow & \mathtt{a}AB\mathtt{e} \\
2 & A & \rightarrow & A\mathtt{bc} \\
3 & & | & \mathtt{b} \\
4 & B & \rightarrow & \mathtt{d}
\end{array}
$$

and the input string `abbcde`

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

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

Handles

What are we trying to find?

A substring α of the tree's upper frontier that

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

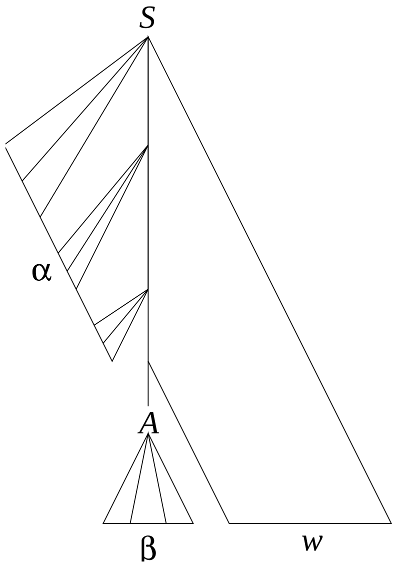
We call such a string a *handle*.

Formally:

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

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

Handles



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

Handles

Theorem:

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

Proof: (by definition)

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

Example

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

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

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

Handle-pruning

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

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

we set i to n and apply the following simple algorithm

for i = n downto 1

1. find the handle $A_i \rightarrow \beta_i$ in γ_i
2. replace β_i with A_i to generate γ_{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 don't have a handle on top of the stack, *shift* an input symbol onto the stack
   b) *prune the handle*
      if we have a handle $A \to \beta$ on the stack, *reduce*
      i) pop $|\beta|$ symbols off the stack
      ii) push A onto the stack

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

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

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

1. *Shift until top of stack is the right end of a handle*
2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

Shift-reduce parsing

Shift-reduce parsers are simple to understand

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal LHS
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

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

LR(k) grammars

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

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

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

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

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

LR(k) grammars

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

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

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

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

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

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

Why study LR grammars?

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

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

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

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

Left versus right recursion

Right Recursion:

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

Left Recursion:

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

Rule of thumb:

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

Parsing review

R. 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.

Dilemmas LL dilemma: pick $A \to b$ or $A \to c$?
LR dilemma: pick $A \to b$ or $B \to b$?