On-line Notes

CS536-S23 Intro to PLs and Compilers

These notes were originally written by Professor Emerita Susan Horwitz and used, maintained, and updated by subsequent CS 536 instructors including Beck Hasti, Drew Davidson, Loris D'Antoni, and Aws Albarghouthi.

---

1. Overview
2. Scanning
3. JLex
4. Context-free Grammars
5. Syntax-directed Translation
6. JavaCUP
7. Parsing
8. Top-down Parsing
9. Syntax-directed Translation for Predictive Parsing
10. Symbol Tables and Static Checks
11. Runtime Environments
12. Parameter Passing
13. Runtime Access to Variables
14. Code Generation
15. Optimization

Contents

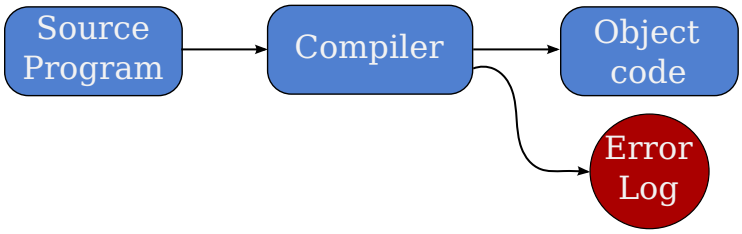
- Introduction
- The Scanner
- The Parser
- The Semantic Analyzer
- The Intermediate Code Generator
- The Optimizer
- The Code Generator

Introduction

What is a compiler?

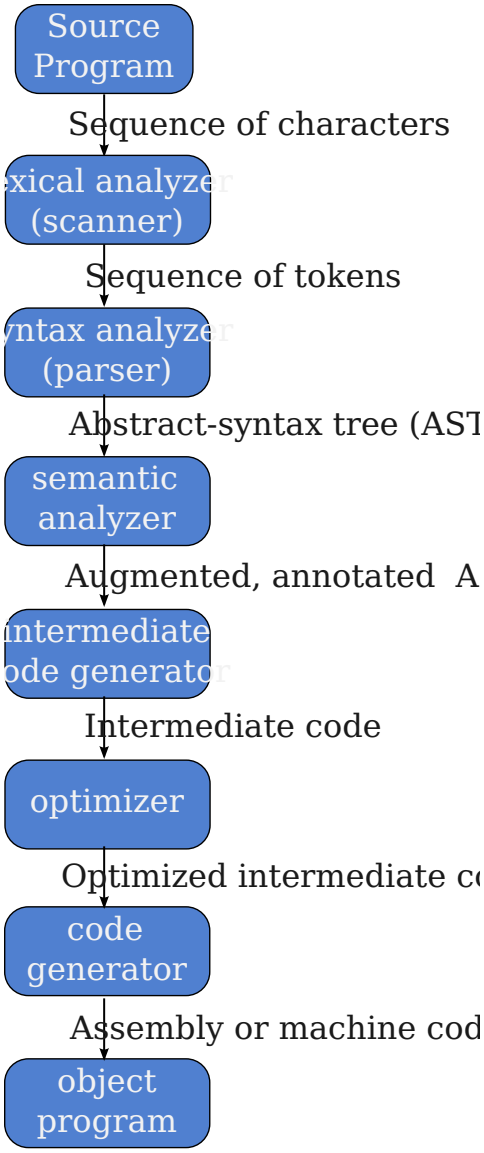
- A recognizer (of some source language L).
- A translator (of programs written in L into programs written in some object or target language L').

Here's a simple pictorial view:



A compiler is *itself* a program, written in some **host** language. (In cs536, students will implement a compiler for a simple source language using Java as the host language.)

A compiler operates in *phases*; each phase translates the source program from one representation to another. Different compilers may include different phases, and/or may order them somewhat differently. A typical organization is shown below.



Below, we look at each phase of the compiler.

The Scanner

The scanner is called by the parser; here's how it works:

- The scanner reads characters from the source program.
- The scanner groups the characters into **lexemes** (sequences of characters that "go together").
- Each lexeme corresponds to a **token**; the scanner returns the next token (plus maybe some additional information) to the parser.
- The scanner may also discover lexical errors (e.g., erroneous characters).

The definitions of what is a lexeme, token, or bad character all depend on the source language.

Example

Here are some Java lexemes and the corresponding tokens:

```
lexeme:                ;          =        index    tmp    37        102

corresponding token:  SEMI-COLON   ASSIGN    IDENT  IDENT  INT-LIT  INT-LIT
```

Note that multiple lexemes can correspond to the same token (e.g., there are many identifiers).

Given the source code:

```
position  =  initial +  rate   *    60 ;
```

a Java scanner would return the following sequence of tokens:

```
IDENT  ASSIGN IDENT PLUS IDENT TIMES INT-LIT SEMI-COLON
```

Erroneous characters for Java source include # and control-a.

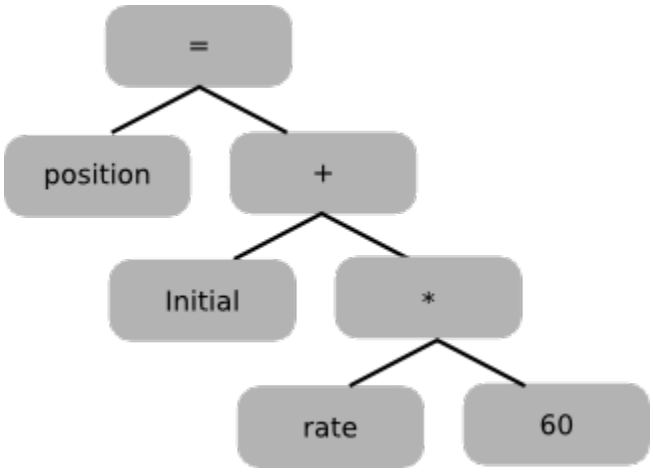
The Parser

- Groups tokens into "grammatical phrases", discovering the underlying structure of the source program.
- Finds syntax errors. For example, in Java the source code
  `position = * 5 ;`
  corresponds to the sequence of tokens:
  `IDENT ASSIGN TIMES INT-LIT SEMI-COLON`
  All are legal tokens, but that sequence of tokens is erroneous.
- Might find some "static semantic" errors, e.g., a use of an undeclared variable, or variables that are multiply declared.
- Might generate code, or build some intermediate representation of the program such as an abstract-syntax tree.

Example

source code: `position = initial + rate * 60 ;`

Abstract syntax tree:



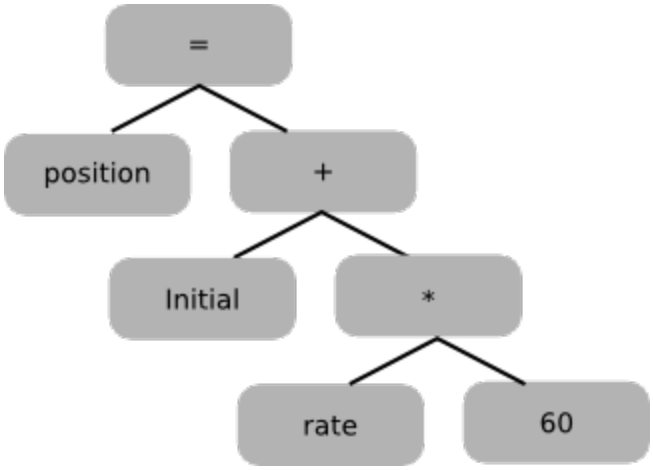
Notes:

- The interior nodes of the tree are **operators**.
- A node's children are its **operands**.
- Each subtree forms a "logical unit", e.g., the subtree with * at its root shows that because multiplication has higher precedence than addition, this operation must be performed as a unit (*not* initial+rate).

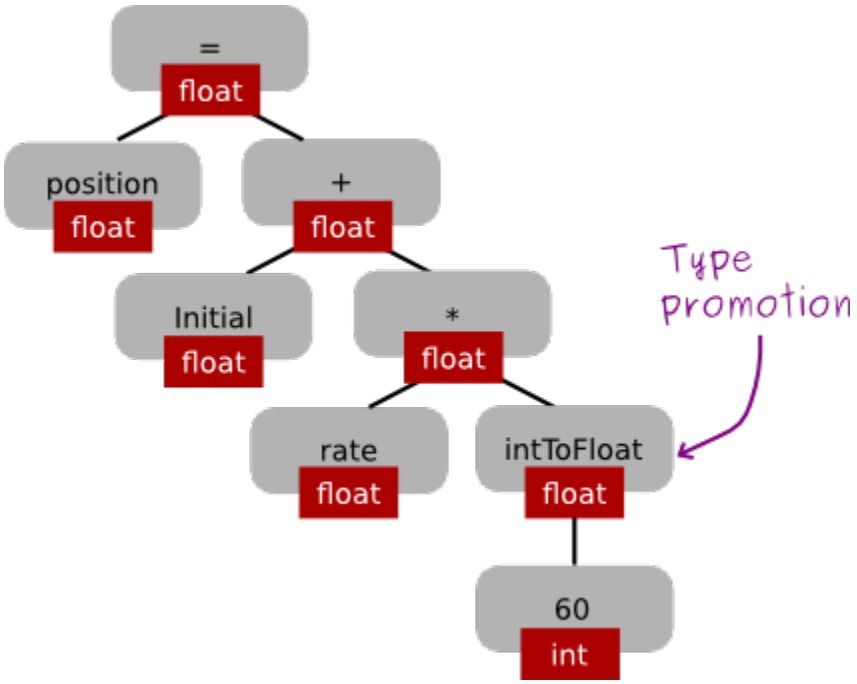
The Semantic Analyzer

The semantic analyzer checks for (more) "static semantic" errors, e.g., type errors. It may also annotate and/or change the abstract syntax tree (e.g., it might annotate each node that represents an expression with its type). **Example:**

Abstract syntax tree before semantic analysis:



Abstract syntax tree after semantic analysis:



The Intermediate Code Generator

The intermediate code generator translates from abstract-syntax tree to intermediate code. One possibility is 3-address code (code in which each instruction involves at most 3 operands). Below is an example of 3-address code for the abstract-syntax tree shown above. Note that in this example, the second and third instructions each have exactly three operands (the location where the result of the operation is stored, and two operators); the first and fourth instructions have just two operands

("temp1" and "60" for instruction 1, and "position" and "temp3" for instruction 4).

```
temp1 = inttofloat(60)
temp2 = rate * temp1
temp3 = initial + temp2
position = temp3
```

The Optimizer

The optimizer tries to improve code generated by the intermediate code generator. The goal is usually to make code run faster, but the optimizer may also try to make the code smaller. In the example above, an optimizer might first discover that the conversion of the integer 60 to a floating-point number can be done at compile time instead of at run time. Then it might discover that there is no need for "temp1" or "temp3". Here's the optimized code:

```
temp2 = rate * 60.0
position = initial + temp2
```

The Code Generator

The code generator generates object code from (optimized) intermediate code. For example, the following code might be generated for our running example:

```
.data
c1:
    .float  60.0
.text
    l.s     $f0,rate
    mul.s   $f0,c1
    l.s     $f2,initial
    add.s   $f0,$f0,$f2
    s.s     $f0,position
```

Contents

- Overview
- Finite-State Machines
  - Example: Pascal Identifiers
  - Test Yourself #1
  - Example: Integer Literals
  - Formal Definition
  - Deterministic and Non-Deterministic FSMs
  - How to Implement a FSM
- Regular Expressions
  - Test Yourself #2
  - Example: Integer Literals
  - The Language Defined by a Regular Expression
- Using Regular Expressions and Finite-State Machines to Define a Scanner
  - Scanning: Problem Definition
  - Method 1
  - Test Yourself #3
  - Method 2

Overview

Recall that the job of the scanner is to translate the sequence of characters that is the input to the compiler to a corresponding sequence of **tokens**. In particular, each time the scanner is called it should find the *longest* sequence of characters in the input, starting with the current character, that corresponds to a token, and should return that token.

It is possible to write a scanner from scratch, but a more efficient and less error-prone approach is to use a **scanner generator** like lex or flex (which produce C code), or JLex (which produces Java code). The input to a scanner generator includes one **regular expression** for each token (and for each construct that must be recognized and ignored, such as whitespace and comments). Therefore, to use a scanner generator you need to understand regular expressions. To understand how the

scanner generator produces code that correctly recognizes the strings defined by the regular expressions, you need to understand **finite-state machines (FSMs)**.

Finite-State Machines

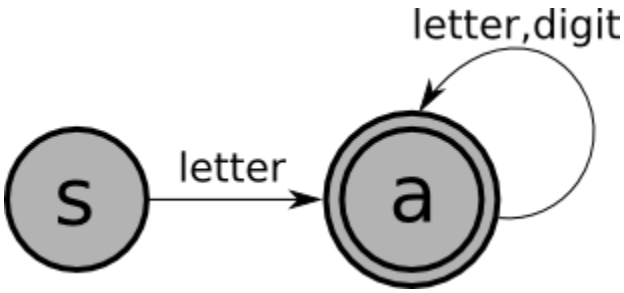
A finite-state machine is similar to a compiler in that:

- A compiler recognizes legal *programs* in some (source) language.
- A finite-state machine recognizes legal *strings* in some language.

In both cases, the input (the program or the string) is a sequence of characters.

Example: Pascal Identifiers

Here's an example of a finite-state-machine that recognizes Pascal identifiers (sequences of one or more letters or digits, starting with a letter):



In this picture:

- Nodes are *states*.
- Edges (arrows) are *transitions*. Each edge should be labeled with a single character. In this example, we've used a single edge labeled "letter" to stand for 52 edges labeled 'a', 'b', ..., 'z', 'A', ..., 'Z'. (Similarly, the label "letter,digit" stands for 62 edges labeled 'a',...'Z','0',...'9'.)
- S is the *start state*; every FSM has exactly one (a standard convention is to label the start state "S").
- A is a *final state*. By convention, final states are drawn using a double circle,

and non-final states are drawn using single circles. A FSM may have more than one final state.

A FSM is applied to an input (a sequence of characters). It either accepts or rejects that input. Here's how the FSM works:

- The FSM starts in its start state.
- If there is a edge out of the current state whose label matches the current input character, then the FSM moves to the state pointed to by that edge, and "consumes" that character; otherwise, it gets stuck.
- The finite-state machine stops when it gets stuck or when it has consumed all of the input characters.

An input string is *accepted* by a FSM if:

- The entire string is consumed (the machine did not get stuck), and
- the machine ends in a final state.

The *language defined by a FSM* is the set of strings accepted by the FSM.

The following strings *are* in the language of the FSM shown above:

- `x`
- `tmp2`
- `XyZzy`
- `position27`

The following strings are *not* in the language of the FSM shown above:

- `123`
- `a?`
- `13apples`

TEST YOURSELF #1

Write a finite-state machine that accepts e-mail addresses, defined as follows:

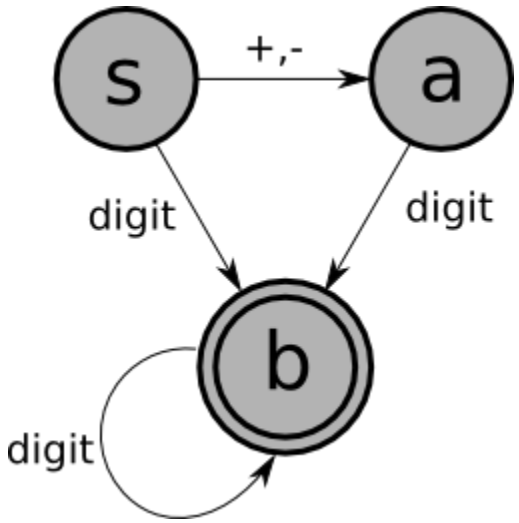
- a sequence of one or more letters and/or digits,

- followed by an at-sign,
- followed by one or more letters,
- followed by zero or more extensions.
- An extension is a dot followed by one or more letters.

solution

Example: Integer Literals

The following is a finite-state machine that accepts integer literals with an optional + or - sign:



Formal Definition

An FSM is a 5-tuple: (Q, Σ, δ, q, F)

- Q is a finite set of states ({S, A, B} in the above example).
- Σ (an uppercase sigma) is the alphabet of the machine, a finite set of characters that label the edges ({+, −, 0, 1, . . . , 9} in the above example).
- q is the start state, an element of Q (S in the above example).
- F is the set of final states, a subset of Q ({B} in the above example).
- δ is the state transition relation: $Q \times \Sigma \to Q$ (i.e., it is a function that takes two arguments -- a state in Q and a character in Σ -- and returns a state in Q).

Here's a definition of δ for the above example, using a **state transition table**:

	+	-	digit
S	A	A	B
A			B
B			B

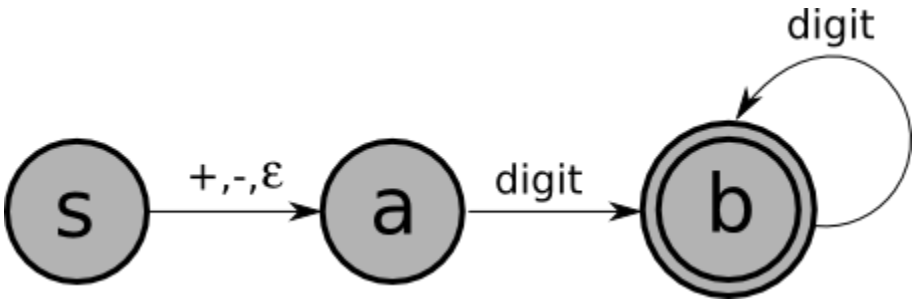
Deterministic and Non-Deterministic FSMs

There are two kinds of FSM:

1. Deterministic:
   - No state has more than one outgoing edge with the same label.
2. Non-Deterministic:
   - States *may* have more than one outgoing edge with same label.
   - Edges may be labeled with ε (epsilon), the empty string. The FSM can take an ε-transition *without* looking at the current input character.

Example

Here is a non-deterministic finite-state machine that recognizes the same language as the second example deterministic FSM above (the language of integer literals with an optional sign):



Sometimes, non-deterministic machines are simpler than deterministic ones, though not in this example.

A string is accepted by a non-deterministic finite-state machine if there exists a

sequence of moves starting in the start state, ending in a final state, that consumes the entire string. For example, here's what happens when the above machine is run on the input "+75":

After scanning	Can be in these states	
(nothing)	S	A
+	A	(stuck)
+7	B	(stuck)
+75	B	(stuck)

There *is* one sequence of moves that consumes the entire input and ends in a final state (state B), so this input *is* accepted by his machine.

It is worth noting that there is a theorem that says:

> For every non-deterministic finite-state machine M, there exists a *deterministic* machine *[Math Processing Error]* such that M and *[Math Processing Error]* accept the *same* language.

How to Implement a FSM

The most straightforward way to *program* a (deterministic) finite-state machine is to use a **table-driven** approach. This approach uses a table with one row for each state in the machine, and one column for each possible character. `Table[j][k]` tells which state to go to from state j on character k. (An empty entry corresponds to the machine getting stuck, which means that the input should be rejected.)

Recall the table for the (deterministic) "integer literal" FSM given above:

	+	-	digit
S	A	A	B
A			B
B			B

The table-driven program for a FSM works as follows:

- Have a variable named state, initialized to S (the start state).
- Repeat:
  - read the next character from the input
  - use the table to assign a new value to the state variable

  until the machine gets stuck (the table entry is empty) or the entire input is read. If the machine gets stuck, reject the input. Otherwise, if the current state is a *final* state, accept the input; otherwise, reject it.

Regular Expressions

Regular expressions provide a compact way to define a language that can be accepted by a finite-state machine. Regular expressions are used in the input to a scanner generator to define each token, and to define things like whitespace and comments that do not correspond to tokens, but must be recognized and ignored.

As an example, recall that a Pascal identifier consists of a letter, followed by zero or more letters or digits. The regular expression for the language of Pascal identifiers is:

letter . (letter | digit)*

The following table explains the symbols used in this regular expression:

|	means "or"
.	means "followed by"
*	means zero or more instances of
( )	are used for grouping

Often, the "followed by" dot is omitted, and just writing two things next to each other means that one follows the other. For example:

letter (letter | digit)*

In fact, the operands of a regular expression should be single characters or the special character epsilon, meaning the empty string (just as the labels on the edges of a FSM should be single characters or epsilon). In the above example, "letter" is used as a shorthand for:

a | b | c | ... | z | A | ... | Z

and similarly for "digit". Also, we sometimes put the characters in quotes (this is necessary if you want to use a vertical bar, a dot, or a star character).

To understand a regular expression, it is necessary to know the precedences of the three operators. They can be understood by analogy with the arithmetic operators for addition, multiplication, and exponentiation:

Regular Expression Operator	**Analogous Arithmetic Operator**	**Precedence**
|	plus	lowest precedence
.	times	middle
*	exponentiation	highest precedence

So, for example, the regular expression:

letter. letter|digit*

does *not* define the same language as the expression given above. Since the dot operator has higher precedence than the | operator (and the * operator has the highest precedence of all), this expression is the same as:

(letter. letter)|(digit*)

and it means "either two letters, or zero or more digits".

TEST YOURSELF #2

Describe (in English) the language defined by each of the following regular expressions:

1. digit|letter letter
2. digit|letter letter*
3. digit|letter*

solution

Example: Integer Literals

An integer literal with an optional sign can be defined in English as:

> "(nothing or + or -) followed by one or more digits"

The corresponding regular expression is:

$$(+|-|\varepsilon).(\text{digit}.\text{digit}^*)$$

Note that the regular expression for "one or more digits" is:

$$\text{digit}.\text{digit}^*$$

i.e., "one digit followed by zero or more digits". Since "one or more" is a common pattern, another operator, +, has been defined to mean "one or more". For example,

$$\text{digit}+$$

means "one or more digits", so another way to define integer literals with optional sign is:

$$(+|-|\varepsilon).\text{digit}+$$

The Language Defined by a Regular Expression

Every regular expression defines a language: the set of strings that match the expression. We will not give a formal definition here, instead, we'll give some

examples:

Regular Expression	Corresponding Set of Strings
ε	{""}
a	{"a"}
a.b.c	{"abc"}
a | b | c	{"a", "b", "c"}
(a | b | c)*	{"", "a", "b", "c", "aa", "ab", ..., "bccabb" ...}

Using Regular Expressions and FSMs to Define a Scanner

There is a theorem that says that for every regular expression, there is a finite-state machine that defines the same language, and vice versa. This is relevant to scanning because it is usually easy to define the tokens of a language using regular expressions, and then those regular expression can be converted to finite-state machines (which can actually be programmed).

For example, let's consider a very simple language: the language of assignment statements in which the left-hand side is a Pascal identifier (a letter followed by one or more letters or digits), and the right-hand side is one of the following:

- ID + ID
- ID * ID
- ID == ID

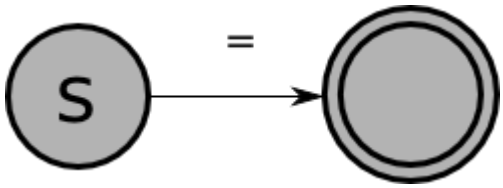
This language has five tokens, which can be defined by the following five regular expressions:

Token	Regular Expression
ASSIGN	"="
ID	letter (letter | digit)*
PLUS	+

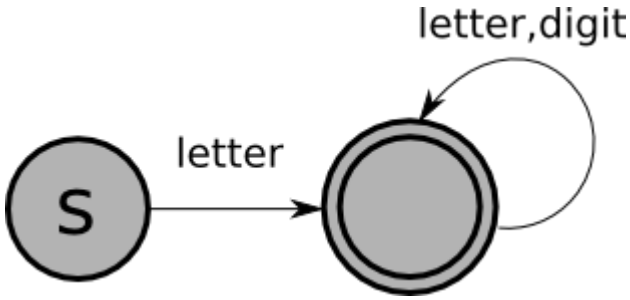
TIMES	*
EQUALS	"="."="

These regular expressions can be converted into the following finite-state machines:

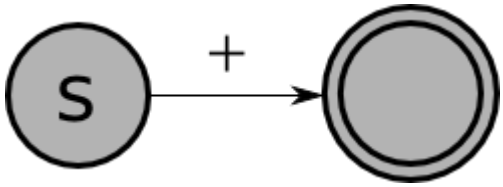
ASSIGN:



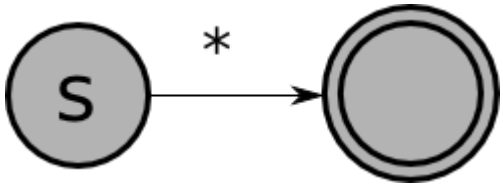
ID:

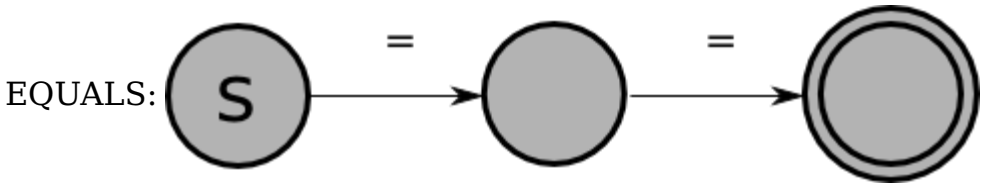


PLUS:



TIMES:





The remainder of this section addresses the following problem: ``Given an FSM for each token, how do we create a scanner?''

Scanning: Problem Definition

An FSM only checks *language membership*. That is, given an FSM M, it can answer the question ``Given a string ω, is ω ∈ L(M)?'' A scanner (a.k.a. a tokenizer) needs more:

- It needs to break up into tokens a stream made up of many different tokens (each defined by its own FSM)
- It needs to successively find the *next* token by a ``maximal munch'':
  - the *longest* prefix of the remaining input that corresponds to a token

  and return information about what was matched

Thus, the problem definition is as follows:

> Given a collection of token definitions (in the form of one FSM for each kind of token), create a maximal-munch tokenizer.

Method 1

Recall that the goal of a scanner is to find the *longest prefix* of the current input that corresponds to a token. This has two consequences:

1. The scanner sometimes needs to look one or more characters *beyond* the last character of the current token, and then needs to "put back" those characters so that the next time the

scanner is called it will have the correct current character. For example, when scanning a program written in the simple assignment-statement language defined above, if the input is "==", the scanner should return the EQUALS token, not two ASSIGN tokens. So if the current character is "=", the scanner must look at the next character to see whether it is another "=" (in which case it will return EQUALS), or is some other character (in which case it will put that character back and return ASSIGN).

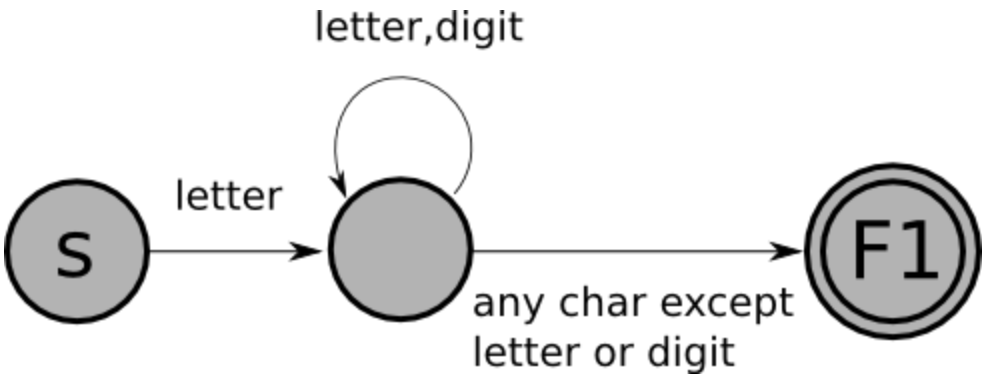
2. It is no longer correct to run the FSM program until the machine gets stuck or end-of-input is reached, since in general the input will correspond to *many* tokens, not just a single token.

Furthermore, remember that regular expressions are used both to define tokens and to define things that must be recognized and skipped (like whitespace and comments). In the first case a value (the current token) must be returned when the regular expression is matched, but in the second case the scanner should simply start up again trying to match another regular expression.

With all this in mind, to create a scanner from a set of FSMs, we must:

- *modify* the machines so that a state can have an associated **action** to "put back N characters" and/or to "return token XXX",
- we must *combine* the finite-state machines for all of the tokens in to a *single* machine, and
- we must write a program for the "combined" machine.

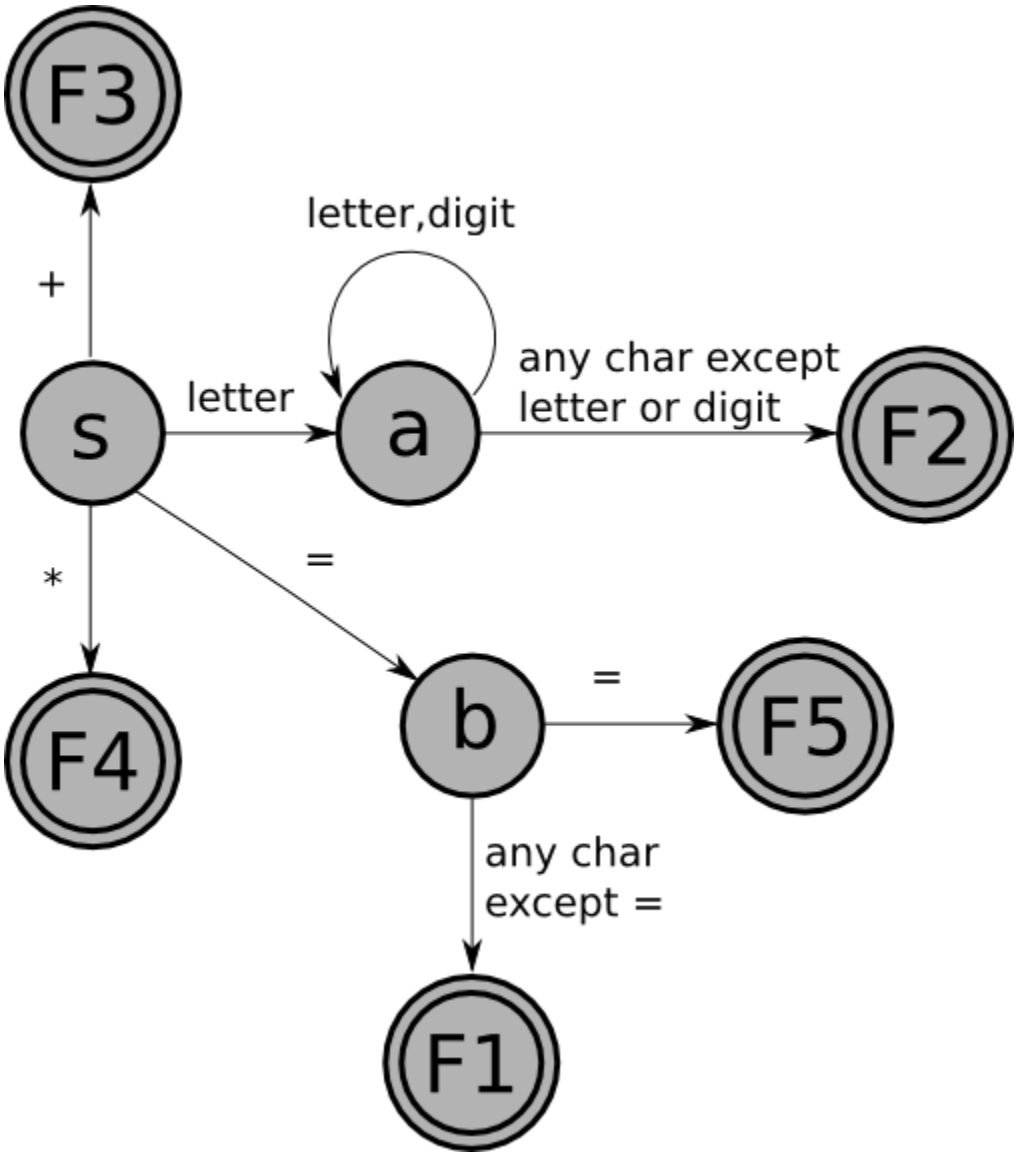
For example, the FSM that recognizes Pascal identifiers must be modified as follows:



with the following table of actions:

Actions:
F1: put back 1 char, return ID

And here is the combined FSM for the five tokens (with the actions noted below):



with the following table of actions:

Actions:

F1: put back 1 char; return ASSIGN
F2: put back 1 char; return ID
F3: return PLUS
F4: return TIMES
F5: return EQUALS

We can convert this FSM to code using the table-driven technique described above, with a few small modifications:

- The table will include a column for end-of-file as well as for all possible characters (the end-of-file column is needed, for example, so that the scanner works correctly when an identifier is the last token in the input).
- Each table entry may include an action as well as or instead of a new state.
- Instead of repeating "read a character; update the state variable" until the machine gets stuck or the entire input is read, the code will repeat: "read a character; perform the action and/or update the state variable" (eventually, the action will be to return a value, so the scanner code will stop, and will start again in the start state next time it is called).

Here's the table for the above "combined" FSM:

	+	*	**=**	letter	digit	**EOF**
S	return PLUS	return TIMES	B	A		
A	put back 1 char; return ID	put back 1 char; return ID	put back 1 char; return ID	A	A	return ID
B	put back 1 char; return	put back 1 char; return	return EQUALS	put back 1 char; return	put back 1 char; return	return ASSIGN

TEST YOURSELF #3

Suppose we want to extend the very simple language of assignment statements defined above to allow both integer and double literals to occur on the right-hand sides of the assignments. For example:

```
x = 23 + 5.5
```

would be a legal assignment.

What new tokens would have to be defined? What are the regular expressions, the finite-state machines, and the modified finite-state machines that define them? How would the the "combined" finite-state machine given above have to be augmented?

solution

Method 2

Unfortunately, the technique for creating a scanner from a set of FSMs described in Method 1 has some drawbacks. As we saw above, the issue that complicates matters has to do with overlaps in tokens:

- = vs. ==
- + vs. +=
- The keyword ``for'' vs. the identifier ``formula''

The scanner must know how to resolve such ambiguities.

In fact, the above examples are all handled correctly by Method 1 (why?), and typically there is no issue for the kind of overlaps that arise in the lexical syntax of a programming language. However, in general, there can be a problem. For instance, consider the following token definitions

Token	Regular Expression
TOKEN1	abc
TOKEN2	(abc)*d

and the input string ``abcabcabc''. The desired result is that the input string should be tokenized as TOKEN1 TOKEN1 TOKEN1.

More generally, suppose the input string were of the form $(abc)^n$, where the superscript ``n'' means that the string has n repetitions of ``abc''. The desired result is that the input string should be tokenized as $TOKEN1^n$. The problem is that for the scanner to establish that the first three characters should be tokenized as TOKEN1—as opposed to making up the first three characters of a longer TOKEN2—the scanner has to visit *all* the characters of the input before deciding that there is no final ``d'' as required for the token to be TOKEN2; consequently, the scanner has to back up 3*(n-1) characters so that the input that remains to be tokenized is $(abc)^{n-1}$. In this case, the amount of backup is proportional to the length of the string, and hence is *unbounded* (i.e., not bounded by any fixed constant, independent of n).

The idea behind the tokenization algorithm is as follows:

- Use one DFA for each kind of token (e.g., M1 for abc and M2 for (abc)*d).
- Start running all DFAs simultaneously on the remaining input.
- A DFA drops out when it enters a stuck state (i.e., has no available transition on the next input character).
- Update most_recent_accepted_position and most_recent_accepted_token whenever any machine enters a final state. (Break ties by assigning some precedence order to the DFAs.)
- When the last DFA drops out,
  - Return most_recent_accepted_token (or FAIL, if most_recent_accepted_token

was never set).
    - For finding the next token, the remaining input starts at most_recent_accepted_position.

Using the *most-recent* accepted position ⇒ the *longest* token is identified ⇒ a ``*maximal* munch'' is performed each time a token is identified.

Example:

Let's consider again the example in which we have TOKEN1 $=_{def}$ abc; TOKEN2 $=_{def}$ (abc)*d; and the input string is ``abcabcabc''. (We've specified TOKEN1 and TOKEN2 using regular expressions, but it is easy to give equivalent FSMs for them. Call them M1 and M2, respectively.) Here is a synopsis of what happens when the tokenization algorithm is run:

- The algorithm consumes the first instance of ``abc.''
    - The machines for both TOKEN1 (M1) and TOKEN2 (M2) are still in play.
    - *M1 is in an accepting state*.
- On the next ``a,'' M1 drops out; M2 is still in play, but it is not in an accepting state.
- After the next ``bcabc,'' M2 drops out, but *it never entered its accepting state*.
- TOKEN1 is returned.
- The remaining input (i.e., abcabc) is handled similarly, and two more instances of TOKEN1 are returned.
- The overall result is that ``abcabcabc'' is tokenized as TOKEN1 TOKEN1 TOKEN1, as desired.

The drawback of the tokenization algorithm is that for an example like the one discussed above, the cost of the algorithm is $O(n^2)$. However, it is possible to give a linear-time algorithm for maximal-munch tokenization. See Reps, T., ```Maximal-munch' tokenization in linear time.'' *ACM TOPLAS 20*, 2 (March 1998), pp. 259-273.

Here is another variant of the tokenization algorithm, which uses *one* DFA. Suppose that the tokens are defined by the regular expressions R_1, R_2, ..., R_k. Let M be a DFA for which $L(M) = L(R_1 \mid R_2 \mid ... \mid R_k)$.

Notation: We use ``mrap'' to abbreviate ``most_recent_accepted_position,'' and ``mrat'' to abbreviate ``most_recent_accepted_token.'' We also assume that there is an auxiliary function, ``tokenFor(q),'' which, for each final state q ∈ F, provides information about what token should be returned when q is the final state corresponding for the most-recent accepted token. It is easy to construct tokenFor(q) during the construction of M, and it is how M accounts for the precedence order among tokens if there is any ambiguity among the token definitions.

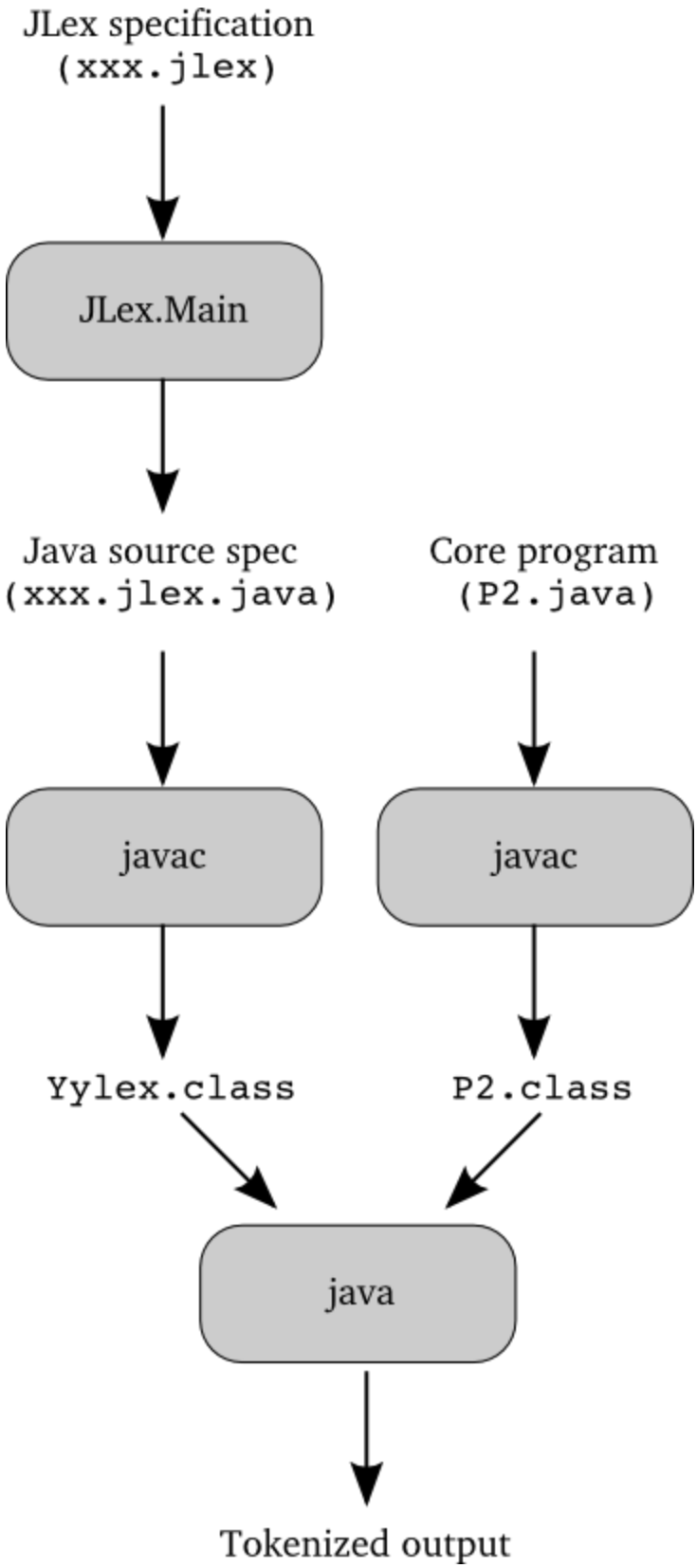
```
Tokenize(M: DFA, input: string)
let [Q, Σ, δ, q0, F] = M in
begin
  i = 0;
  forever {
    q = q0;
    mrap = -1;
    mrat = -1;
    while (i < length(input)) {
      q = δ(q, input[i]);
      i = i + 1;
      if (q ∈ F) {
        mrap = i;
        mrat = tokenFor(q);
      }
    }
    if (mrap == -1) return 'FAIL'
    i = mrap;
    print(mrat);
    if (i ≥ length(input)) return 'SUCCESS'
  }
end
```

Contents

- Overview
- Format of a JLex Specification
  - Regular Expression Rules
    - Test Yourself #1
  - JLex Directives
    - Test Yourself #2
  - Comments
  - States
    - Test Yourself #3
- yyline and yytext
- A Small Example
- Quick Reference Guide

Overview

JLex is a scanner generator that produces Java code. Here's a picture illustrating how to create and run a program using JLex:



The input to JLex is a specification that includes a set of regular expressions and associated actions. The output of JLex is a Java source file that defines a class named **Yylex**. Yylex includes a constructor that is called with one argument: the input stream

(an InputStream or a Reader). It also includes a method called **next_token**, which returns the next token in the input.

The picture above assumes that a class named P2 has been defined that contains the core program of interest. That program will declare an object of type Yylex, and will include calls to the Yylex constructor and its next_token method.

Format of a JLex Specification

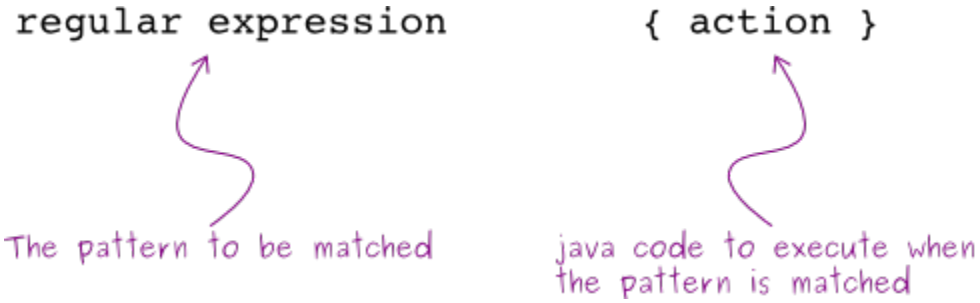
A JLex specification has three parts, separated by double percent signs:

1. **User code**: this part of the specification will not be discussed here.
2. **JLex directives**: This includes macro definitions (described below). See the JLex Reference Manual for more information about this part of the specification.
3. **Regular expression rules**: These rules specify how to divide up the input into tokens. Each rule includes an optional state list, a regular expression, and an associated action.

We will discuss the regular expression rules part first.

Regular Expressions Rules

The state-list part of a rule is discussed below. Ignoring state-lists for now, the form of a regular expression rule is:



When the scanner's next_token method is called, it repeats:

1. Find the longest sequence of characters in the input (starting with the current character) that matches a regular-expression pattern.
2. Perform the associated action.

until an action causes the next_token method to return. If there are several patterns that match the same (longest) sequence of characters, then the first such pattern is considered to be matched (so the order of the regular-expression rules can be important).

If an input character is not matched in any pattern, the scanner throws an exception. It is *not* good to have a scanner that can "crash" on bad input, so it is important to make sure that there can be no such unmatched characters!

The regular expressions are similar to the ones discussed in the scanner notes. Here's how they are used to match the input:

- Most characters match themselves. For example:
  - `abc`
  - `==`
  - `while`

  are three patterns that match exactly those sequences of characters (note that writing one character after another means "followed by" as usual).

- Characters (even special characters, except backslash) enclosed in double quotes match themselves. For example, the following patterns are equivalent to the three given above:
  - `"abc"`
  - `"=="`
  - `"while"`

    And the following pattern:

```
"a|b"
```

matches the three-character sequence: `a` then | then `b`, rather than matching a single `a` or a single `b`.

- The following characters have the usual special meanings as regular expression operators:

|	means "or"
*	means zero or more instances of
+	means one or more instances of
?	means zero or one instance of
( )	are used for grouping

- The dot character matches *any* character except the newline character. It is usually used in the last rule in the specification, to match all "bad" characters (and the associated action issues an error message).

- The backslash is a special *escape* character:

\n	newline
\t	tab
\"	double quote

To match a backslash character, put *two* backslashes in a character class (see below). See the JLex Reference Manual for a complete list of the special characters escaped by a backslash.

- The carat and dollar-sign characters: `^` and `$`, are special characters. When the carat is used as the first character in a pattern, it causes the pattern to match only at the beginning of a line (i.e., only if the previous character was a newline). When the dollar sign is used as the last character in a pattern, it causes the pattern to match only at the end of a line (i.e., only if the next character is a newline).

- The regular expression can include **character classes**, delimited by square brackets:
  - A character class will match *one* character.
  - If no special characters are used inside the character class, then the character class matches any of the characters it includes inside its square brackets. For example: `[abc]` matches an `a`, or a `b`, or a `c`, so it is the same as: `a|b|c`.
  - Here are the characters that are "special" inside a character class:

-	means a range of characters; e.g., `a-z` means "a to z".
^	is only a special character if it is the *first* character in the square brackets; it means *not* any of the following characters. So for example, `[^abc]` matches any character other than an `a`, or a `b`, or a `c`.
\	is used as an escape character with `n`, `b`, `"`, etc as usual; it can also be used to escape the characters that are special inside a character class (e.g., `[a\-z]` matches an `a` or a `-` or a `z`, and `[\\]` matches a backslash.
"	can be used around characters that are special inside a character class to make them match themselves (e.g., `["\"]` matches a backslash, and `["-"]` matches a hyphen. To include a double-quote character in a character class, escape it with a backslash.

Note that whitespace only matches itself if it is inside quotes or in a character class; otherwise, it *ends* the current pattern. So the two rules:

```
[a bc]    {}
a|" "|b|c {}
```

are equivalent; each matches an `a`, or a space, or a `b`, or a `c`, while the rule:

```
a bc   {}
```

causes an error when you try to process your specification.

TEST YOURSELF #1

Question 1: The character class `[a-zA-Z]` matches any letter. Write a character class that matches any letter or any digit.

Question 2: Write a pattern that matches any Pascal identifier (a sequence of one or more letters and/or digits, starting with a letter).

Question 3: Write a pattern that matches any C identifier (a sequence of one or more letters and/or digits and/or underscores, starting with a letter or underscore).

Question 4: Write a pattern that matches any C identifier that does not *end* with an underscore.

solution

JLex directives

Recall that the second part of a JLex specification contains directives. This can include specifying the value that should be returned on end-of-file, specifying that line counting should be turned on, and specifying that the scanner will be used with the Java parser generator java cup. (See the JLex Reference Manual for more information about directives.)

The directives part also includes **macro definitions**. The form of a macro definition is:

```
name = regular-expression
```

where `name` is any valid Java identifier, and `regular-expression` is any regular expression as defined above. Here are some examples:

```
DIGIT =         [0-9]
LETTER =        [a-zA-Z]
WHITESPACE =    [ \t\n]
```

Once a macro has been defined, it can be used in a regular expression (either to define another macro, or in the "Regular Expression Rules" part of the JLex specification. To use a macro, just use its name inside curly braces. For example, given the above macro definitions, the following pattern could be used to match Pascal identifiers:

```
{LETTER}({LETTER}|{DIGIT})*
```

TEST YOURSELF #2

Define a macro named NOTSPECIAL that matches any character except a newline, double quote, or backslash.

solution

Comments

You can include comments in the first and second parts of your JLex specification, but not in the third part (because JLex will think they are part of a pattern). JLex comments are like Java comments: they start with two slashes, and continue to the end of the line.

States

Recall that each regular expression rule (a pattern and the action to be performed when the pattern is matched) can optionally include a *list of states* at the beginning of the pattern. For example:

```
<STATE1, STATE2>"abc" { }
```
is a rule that starts with a list of two states (named `STATE1` and `STATE2`).

Each time the scanner is called, it is in some state. Initially, it is in a special state called `YYINITIAL`. It will stay in that state unless it matches a pattern whose corresponding action includes code that causes it to change to another state. For example, given the rule:

```
"xyz" { yybegin(STATE1); }
```
if the input contains "xyz", then the call to `yybegin` will be executed, and the scanner will enter the `STATE1` state.

If a rule has *no* list of states, then it will be matched in any state; however, if it has a list of states, then it will be matched only when the scanner is in one of those states. So for example, the rule for "abc" given above will only be matched after the rule for "xyz" has been matched.

Every state other than `YYINITIAL` must be declared in the JLex directives part of the JLex specification. Here's an example declaration:

```
%state STATE1
```

Suppose that for floating-point numbers you want your scanner to return two values: the value before the decimal point, and the value after the decimal point. Here's an example of using a JLex state to do that (using some pseudo-code):

```
%%

DIGIT= [0-9]
DOT=   "."

%state DOTSTATE

%%

<YYINITIAL>{DIGIT}+{DOT}    { yybegin(DOTSTATE);
                              -- save the value so far --
                            }
```

```
<DOTSTATE>{DIGIT}+          { yybegin(YYINITIAL);
                              -- return the saved value and the new one --
                            }
```

TEST YOURSELF #3

A *quoted string* consists of three parts:

1. A double quote.
2. Some text.
3. A double quote.

The text can contain any characters except a newline or a single double-quote character. It *can* contain an "escaped" quote, which is two double-quote characters in a row.

Use JLex states to write a specification to recognize quoted strings, and to return the number of escaped quotes in each such string. To declare a counter, declare a class with a static, public int field, in the "User Code" part of the JLex specification, and update/return that static field.

solution

---

yyline and yytext

If you turn line counting on (by including `%line` in the "directives" part of the specification), you can use the variable `yyline` in the actions that you write for the regular expressions. The value of `yyline` will be the current line number in the input file, counting from zero (so to use that number in error messages printed by your scanner, you will need to add one to `yyline`).

You can also use the method `yytext()` in your actions. This method returns a String -- the sequence of characters that was just matched.

A Small Example

Here is a small (complete) JLex specification:

```
%%

DIGIT=          [0-9]
LETTER=         [a-zA-Z]
WHITESPACE=     [ \t\n]      // space, tab, newline

// The next 3 lines are included so that we can use the generated scanner
// with java CUP (the Java parser generator)
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

// Turn on line counting
%line

%%

{LETTER}({LETTER}|{DIGIT})* {System.out.println(yyline+1 + ": ID " + yytext());}
{DIGIT}+                    {System.out.println(yyline+1 + ": INT");}
"="                         {System.out.println(yyline+1 + ": ASSIGN");}
"=="                        {System.out.println(yyline+1 + ": EQUALS");}
{WHITESPACE}*               { }
.                           {System.out.println(yyline+1 + ": bad char");}
```

Note that the actions in this example are *not* what you would really put in a JLex specification for a scanner. Instead of printing, the first four actions should return the appropriate tokens.

Quick Reference Guide

Operators and Special Symbols in JLex

The following table summarizes the operators and special symbols used in JLex. Note that some characters have an entirely different meaning when used in a regular expression and in a character class. Character classes are always delimited by square brackets; they can be used in the regular expressions that define macros, as well as in the regular expressions used to specify a pattern to be matched in the input.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
(	Matches with) to group sub-expressions.	Represents itself.
)	Matches with (to group sub-expressions.	Represents itself.
[	Begins a character class.	Represents itself.
]	Is illegal.	Ends a character class.
{	Matches with } to delimit a macro name.	Matches with } to delimit a macro name.
}	Matches with { to delimit a macro name.	Represents itself or matches with { to delimit a macro name.
"	Matches with " to delimit strings (only \ is special within strings).	Matches with " to delimit a string of characters that belong to the character class. Only \" is special within the string.
\	Escapes special characters (n, t, etc). Also used to specify a character by its octal, hexadecimal, or unicode value.	Escapes characters that are special inside a character class.
.	Matches any one character	Represents itself.

	except newline.	
|	Alternation (or) operator.	Represents itself.
*	Kleene closure operator (zero or more matches).	Represents itself.
+	Positive closure operator (one or more matches).	Represents itself.
?	Optional choice operator (zero or one matches).	Represents itself.
^	Matches only at beginning of a line.	When it is the first character in the character class, complements the remaining characters in the class.
$	Matches only at end of a line.	Represents itself.
-	Represents itself.	Range of characters operator.

Contents

- Overview
- Example: Simple Arithmetic Expressions
- Formal Definition
- Example: Boolean Expressions, Assignment Statements, and If Statements
- Test Yourself #1
- The Language Defined by a CFG
  - Leftmost and Rightmost Derivations
  - Parse Trees
  - Test Yourself #2
- Ambiguous Grammars
- Expression Grammars
  - Precedence
  - Associativity
    - Test Yourself #3
  - Test Yourself #4
- List Grammars
- A Grammar for a Programming Language
- Summary

Overview

Recall that the input to the parser is a sequence of tokens (received interactively, via calls to the scanner). The parser:

- Groups the tokens into "grammatical phrases".
- Discovers the underlying structure of the program.
- Finds syntax errors.
- Perhaps also performs some actions to find other kinds of errors.

The output depends on whether the input is a syntactically legal program; if so, then the output is some representation of the program:

- an abstract-syntax tree (maybe + a symbol table),
- or intermediate code,
- or object code.

We know that we can use regular expressions to define languages (for example, the languages of the tokens to be recognized by the scanner). Can we use them to define the language to be recognized by the parser? Unfortunately, the answer is no. Regular

expressions are not powerful enough to define many aspects of a programming language's syntax. For example, a regular expression cannot be used to specify that the parentheses in an expression must be balanced, or that every ``else'' statement has a corresponding ``if''. Furthermore, a regular expression doesn't say anything about underlying *structure*. For example, the following regular expression defines integer arithmetic involving addition, subtraction, multiplication, and division:

```
digit+ (("+" | "-" | "*" | "/") digit+)*
```

but provides no information about the precedence and associativity of the operators.

So to specify the syntax of a programming language, we use a different formalism, called **context-free grammars**.

Example: Simple Arithmetic Expressions

We can write a context-free grammar (CFG) for the language of (very simple) arithmetic expressions involving only subtraction and division. In English:

- An integer is an arithmetic expression.
- If exp_1 and exp_2 are arithmetic expressions, then so are the following:
  - exp_1 - exp_2
  - exp_1 / exp_2
  - (exp_1)

Here is the corresponding CFG:

```
exp → INTLITERAL
exp → exp MINUS exp
exp → exp DIVIDE exp
exp → LPAREN exp RPAREN
```

And here is how to understand the grammar:

- The grammar has five **terminal** symbols: `INTLITERAL MINUS DIVIDE LPAREN RPAREN`. The terminals of a grammar used to define a programming language are the tokens returned by the scanner.

- The grammar has one **nonterminal**: `exp` (note that a single name, `exp`, is used instead of $\texttt{exp}_1$ and $\texttt{exp}_2$ as in the English definition above).
- The grammar has four **productions** or **rules**, each of the form: `exp` → `...` A production left-hand side is a single nonterminal. A production right-hand side is either the special symbol ε (the same ε that can be used in a regular expression) or a sequence of one or more terminals and/or nonterminals (there is no rule with ε on the right-hand side in the example given above).

A more compact way to write this grammar is:

```
exp → INTLITERAL | exp MINUS exp |
exp DIVIDE exp | LPAREN exp RPAREN
```

Intuitively, the vertical bar means ``or'', but do **not** be fooled into thinking that the right-hand sides of grammar rules can contain regular expression operators! This use of the vertical bar is just shorthand for writing multiple rules with the same left-hand-side nonterminal.

Formal Definition

A CFG is a 4-tuple (N, Σ, P, S) where:

- N is a set of nonterminals.
- Σ is a set of terminals.
- P is a set of productions (or rules).
- S is the start nonterminal (sometimes called the goal nonterminal) in N. If not specified, then it is the nonterminal that appears on the left-hand side of the first production.

Example: Boolean Expressions, Assignment Statements, and If Statements

The language of boolean expressions can be defined in English as follows:

- "true" is a boolean expression, recognized by the token `TRUE`.
- "false" is a boolean expression, recognized by the token `FALSE`.

- If exp_1 and exp_2 are boolean expressions, then so are the following:
  - exp_1 || exp_2
  - exp_1 && exp_2
  - ! exp_1
  - (exp_1)

Here is the corresponding CFG:

```
bexp → TRUE
bexp → FALSE
bexp → bexp OR bexp
bexp → bexp AND bexp
bexp → NOT bexp
bexp → LPAREN bexp RPAREN
```

Here is a CFG for a language of very simple assignment statements (only statements that assign a boolean value to an identifier):

```
stmt → ID ASSIGN bexp SEMICOLON
```

We can ``combine'' the two grammars given above, and add two more rules to get a grammar that defines the language of (very simple) if statements. In words, an if statement is:

1. The word "if", followed by a boolean expression in parentheses, followed by a statement, or
2. The word "if", followed by a boolean expression in parentheses, followed by a statement, followed by the word "else", followed by a statement.

And here's the grammar:

```
stmt → IF LPAREN bexp RPAREN stmt
stmt → IF LPAREN bexp RPAREN stmt
ELSE stmt
stmt → ID ASSIGN bexp SEMICOLON
bexp → TRUE
bexp → FALSE
bexp → bexp OR bexp
bexp → bexp AND bexp
bexp → NOT bexp
bexp → LPAREN bexp RPAREN
```

TEST YOURSELF #1

Write a context-free grammar for the language of very simple while loops (in which the loop body only contains one statement) by adding a new production with nonterminal `stmt` on the left-hand side.

solution

The Language Defined by a CFG

The language defined by a context-free grammar is the set of strings (sequences of terminals) that can be **derived** from the start nonterminal. What does it mean to derive something?

- Start by setting the "current sequence" to be the start nonterminal.
- Repeat:
  - find a nonterminal X in the current sequence;
  - find a production in the grammar with X on the left (i.e., of the form X → α, where α is either ε (the empty string) or a sequence of terminals and/or nonterminals);
  - Create a new "current sequence" in which α replaces the X found above;

  until the current sequence contains no nonterminals.

Thus we arrive either at epsilon or at a string of terminals. That is how we derive a string in the language defined by a CFG.

Below is an example derivation, using the 4 productions for the grammar of arithmetic expressions given above. In this derivation, we use the actual lexemes instead of the token names (e.g., we use the symbol "-" instead of MINUS).

> exp ⟶ exp - exp ⟶ 1 - exp ⟶ 1 - exp / exp ⟶ 1 - exp / 2 ⟶ 1 - 4 / 2

And here is some useful notation:

> ⟹ means **derives in one step**

$\overset{+}{\Longrightarrow}$ means **derives in one or more steps**

$\overset{*}{\Longrightarrow}$ means **derives in zero or more steps**

So, given the above example, we could write:

$\texttt{exp} \overset{+}{\Longrightarrow} \texttt{1 - exp / exp}$.

A more formal definition of what it means for a CFG G to define a language may be stated as follows:

$$L(G) = \{w | S \overset{+}{\longrightarrow} w\}$$

where

- S is the start nonterminal of G
- w is a sequence of terminals or ε

Leftmost and Rightmost Derivations

There are several kinds of derivations that are important. A derivation is a **leftmost** derivation if it is always the leftmost nonterminal that is chosen to be replaced. It is a **rightmost** derivation if it is always the rightmost one.

Parse Trees

Another way to derive things using a context-free grammar is to construct a **parse tree** (also called a derivation tree) as follows:

- Start with the start nonterminal.
- Repeat:
  - choose a leaf nonterminal X
  - choose a production X ⟶ α
  - the symbols in α become the children of X in the tree

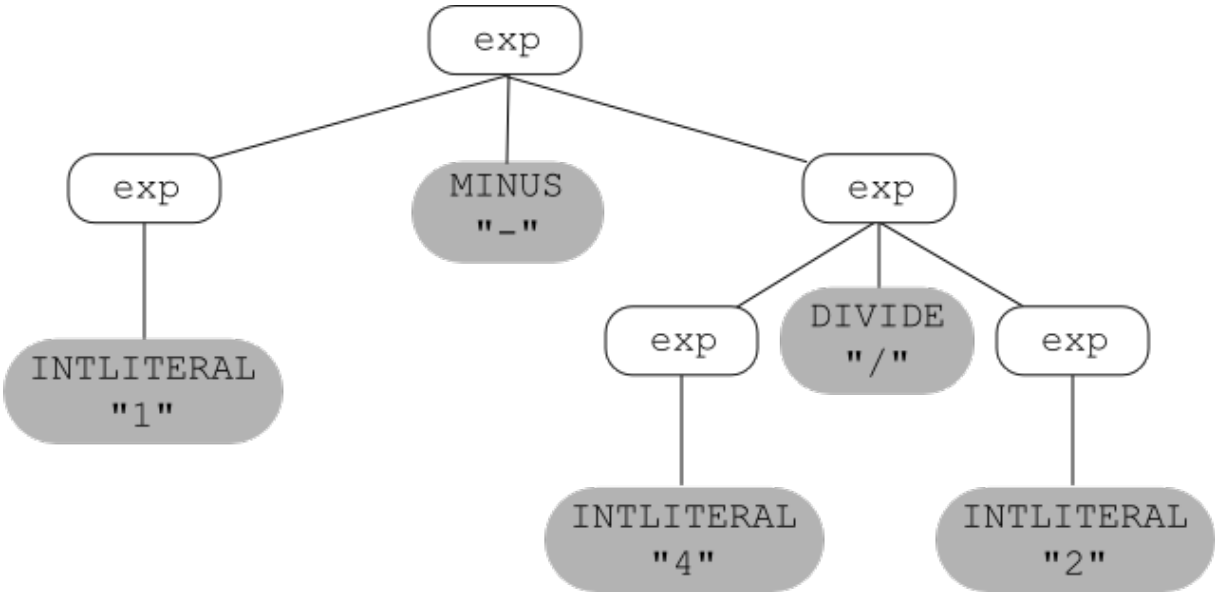
  until there are no more leaf nonterminals left.

The derived string is formed by reading the leaf nodes from left to right.

Here is the example expression grammar given above:

```
exp → INTLITERAL
exp → exp MINUS exp
exp → exp DIVIDE exp
exp → LPAREN exp RPAREN
```

and, using that grammar, here's a parse tree for the string `1 - 4 / 2`:



TEST YOURSELF #2

Below is the CFG for very simple if statements used earlier.

```
stmt → IF LPAREN bexp RPAREN stmt
stmt → IF LPAREN bexp RPAREN stmt
ELSE stmt
stmt → ID ASSIGN bexp SEMICOLON
bexp → TRUE
bexp → FALSE
bexp → bexp OR bexp
bexp → bexp AND bexp
bexp → NOT bexp
bexp → LPAREN bexp RPAREN
```

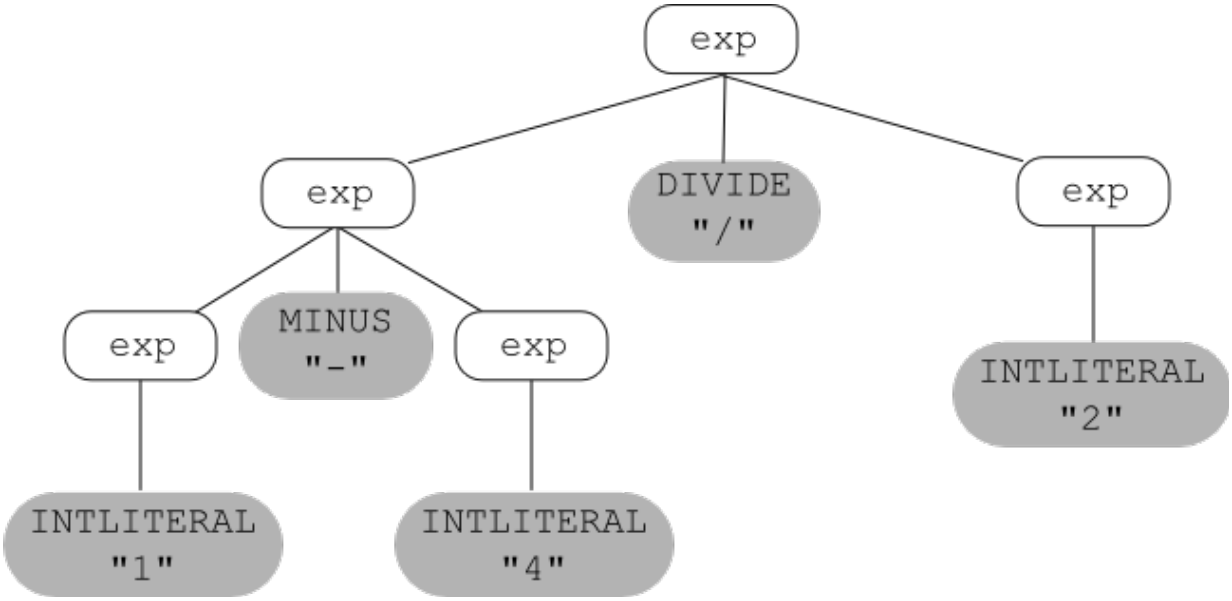
Question 1: Give a derivation for the string: `if (! true ) x = false;` Is your derivation leftmost, rightmost, or neither?

Question 2: Give a parse tree for the same string.

solution

Ambiguous Grammars

The string `1 - 4 / 2` has *two* parse trees using the example expression grammar. One was given above; here's the other one:



If for grammar G and string w there is:

- more than one leftmost derivation of w or,
- more than one rightmost derivation of w, or
- more than one parse tree for w

then G is called an **ambiguous** grammar. (Note: the three conditions given above are equivalent; if one is true then all three are true.)

In general, ambiguous grammars cause problems:

- Ambiguity can make parsing difficult.
- The underlying structure of the language defined by an ambiguous grammar is ill-defined (in the above example, the relative precedences of subtraction and division are not uniquely defined; the first parse tree groups 4/2 while the second groups 1-4, and those two groupings correspond to expressions with *different* values).

Expression Grammars

Since every programming language includes expressions, it is useful to know how to write a grammar for an expression language so that

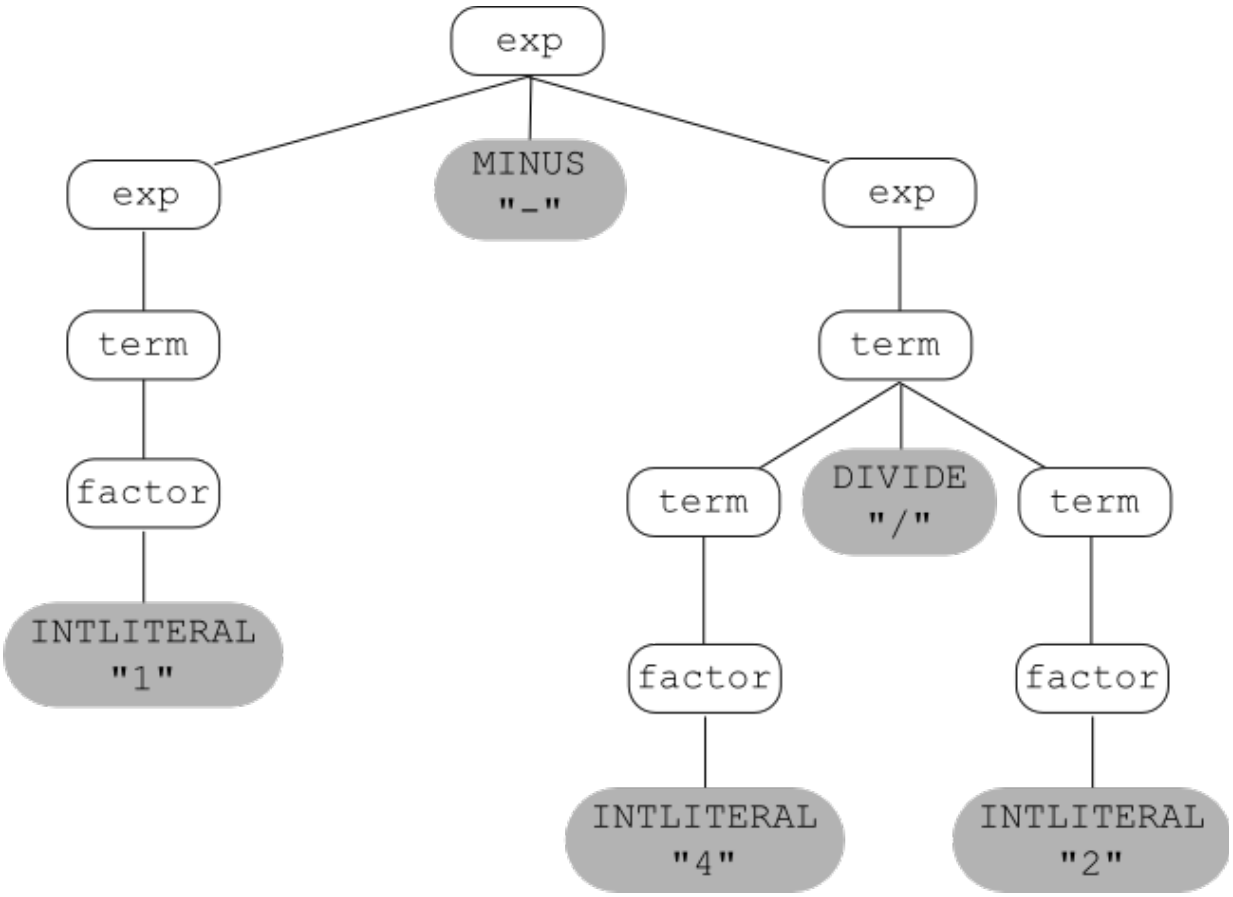
the grammar correctly reflects the precedences and associativities of the operators.

Precedence

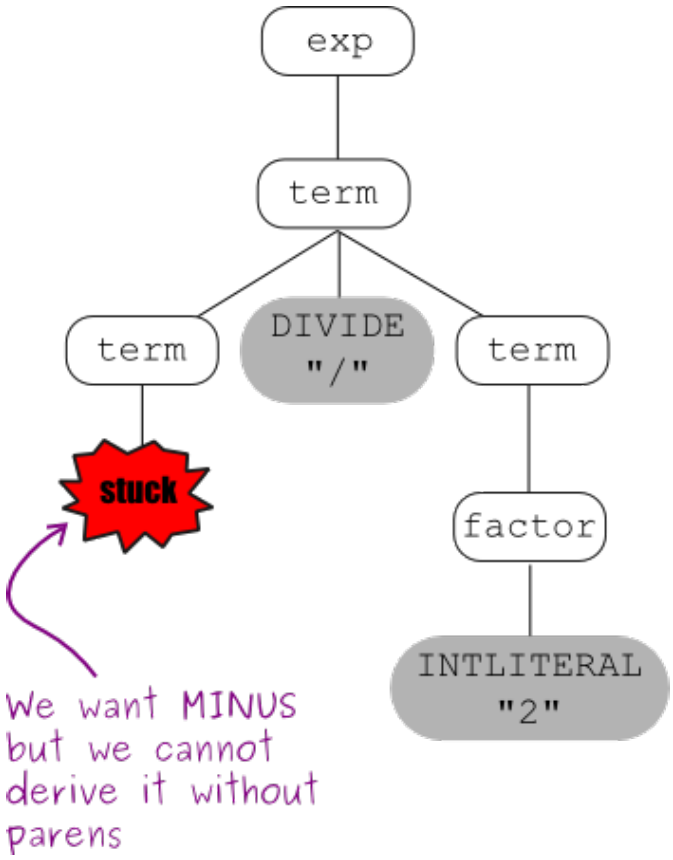
To write a grammar whose parse trees express precedence correctly, use a different nonterminal for each precedence level. Start by writing a rule for the operator(s) with the lowest precedence ("-" in our case), then write a rule for the operator(s) with the next lowest precedence, etc:

```
exp → exp MINUS exp | term
term → term DIVIDE term | factor
factor → INTLITERAL | LPAREN exp
RPAREN
```

Now let's try using these new rules to build parse trees for `1 - 4 / 2`. First, a parse tree that correctly reflects that fact that division has higher precedence than subtraction:



Now we'll try to construct a parse tree that shows the *wrong* precedence:



Associativity

This grammar captures operator precedence, but it is still ambiguous! Parse trees using this grammar may not correctly express the fact that both subtraction and division are *left* associative; e.g., the expression: `5-3-2` is equivalent to: `((5-3)-2)` and *not* to: `(5-(3-2))`.

TEST YOURSELF #3

Draw two parse trees for the expression `5-3-2` using the current expression grammar:

```
exp → exp MINUS exp | term
term → term DIVIDE term | factor
factor → INTLITERAL | LPAREN exp
RPAREN
```

One of your parse trees should correctly group `5-3`, and the other should incorrectly group `3-2`.

solution

To understand how to write expression grammars that correctly reflect the associativity of the operators, you need to understand about **recursion** in grammars.

- A grammar is *recursive in nonterminal* X if:

$$X \overset{+}{\longrightarrow} \ldots X \ldots$$

  (in one or more steps, X derives a sequence of symbols that includes an X).
- A grammar is ***left*** *recursive in* X if:

$$X \overset{+}{\longrightarrow} X \ldots$$

  (in one or more steps, X derives a sequence of symbols that *starts* with an X).
- A grammar is ***right*** *recursive in* X if:

$$X \overset{+}{\longrightarrow} \ldots X$$

  (in one or more steps, X derives a sequence of symbols that *ends* with an X).

The grammar given above for arithmetic expressions is both left and right recursive in nonterminals `exp` and `term` (can you write the derivation steps that show this?).

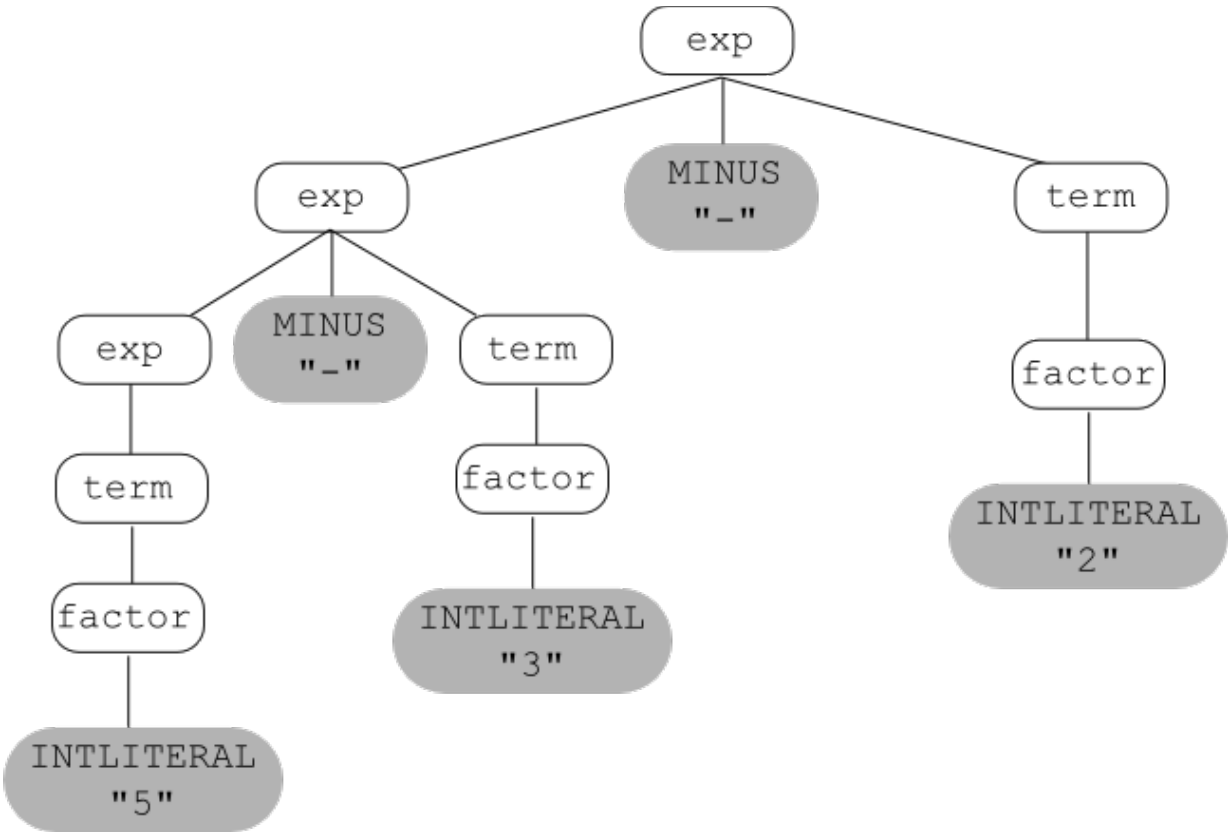
To write a grammar that correctly expresses operator associativity:

- For left associativity, use left recursion.
- For right associativity, use right recursion.

Here's the correct grammar:

```
exp → exp MINUS term | term
term → term DIVIDE factor | factor
factor → INTLITERAL | LPAREN exp
RPAREN
```

And here's the (one and only) parse tree that can be built for `5 - 3 - 2` using this grammar:



Now let's consider a more complete expression grammar, for arithmetic expressions with addition, multiplication, and exponentiation, as well as subtraction and division. We'll use the token POW for the exponentiation operator, and we'll use "**" as the corresponding lexeme; e.g., "two to the third power" would be written: `2 ** 3`, and the corresponding sequence of tokens would be: `INTLITERAL POW INTLITERAL`. Here's an ambiguous context-free grammar for this language:

```
exp → exp PLUS exp | exp MINUS exp |
      exp TIMES exp | exp DIVIDE exp
    | exp POW exp | LPAREN exp RPAREN |
      INTLITERAL
```

First, we'll modify the grammar so that parse trees correctly reflect the fact that addition and subtraction have the same, lowest precedence; multiplication and division have the same, middle precedence; and exponentiation has the highest precedence:

```
exp → exp PLUS exp | exp MINUS exp | term
```

```
term     → term TIMES term | term DIVIDE term | factor
factor   → factor POW factor | exponent
exponent → INTLITERAL | LPAREN exp RPAREN
```

This grammar is still ambiguous; it does not yet reflect the associativities of the operators. So next we'll modify the grammar so that parse trees correctly reflect the fact that all of the operators except exponentiation are left associative (and exponentiation is right associative; e.g., `2**3**4` is equivalent to: `2**(3**4)`):

```
exp      → exp PLUS term | exp MINUS term | term
term     → term TIMES factor | term DIVIDE factor | factor
factor   → exponent POW factor | exponent
exponent → INTLITERAL | LPAREN exp RPAREN
```

Finally, we'll modify the grammar by adding a **unary** operator, unary minus, which has the highest precedence of all (e.g., `-3**4` is equivalent to: `(-3)**4`, not to `-(3**4)`. Note that the notion of associativity does not apply to unary operators, since associativity only comes into play in an expression of the form: `x op y op z`.

```
exp      → exp PLUS term | exp MINUS term | term
term     → term TIMES factor | term DIVIDE factor | factor
factor   → exponent POW factor | exponent
```

```
exponent → MINUS exponent | final

final    → INTLITERAL | LPAREN exp RPAREN
```

TEST YOURSELF #4

Below is the grammar we used earlier for the language of boolean expressions, with two possible operands: `true false`, and three possible operators: `and or not`:

```
bexp → TRUE
bexp → FALSE
bexp → bexp OR bexp
bexp → bexp AND bexp
bexp → NOT bexp
bexp → LPAREN bexp RPAREN
```

Question 1: Add nonterminals so that `or` has lowest precedence, then `and`, then `not`. Then change the grammar to reflect the fact that both `and` and `or` are left associative.

Question 2: Draw a parse tree (using your final grammar for Question 1) for the expression: `true and not true`.

solution

List Grammars

Another kind of grammar that you will often need to write is a grammar that defines a **list** of something. There are several common forms. For each form given below, we provide three different grammars that define the specified list language.

- One or more `PLUS`es (without any separator or terminator). (Remember, *any* of the following three grammars defines this language; you don't need all three lines).
  1. *xList* ⟶ `PLUS` | *xList xList*
  2. *xList* ⟶ `PLUS` | *xList* `PLUS`
  3. *xList* ⟶ `PLUS` | `PLUS` *xList*
- One or more runs of one or more `PLUS`es,

each run separated by commas:
    1. *xList* ⟶ PLUS | *xList* COMMA *xList*
    2. *xList* ⟶ PLUS | *xList* COMMA PLUS
    3. *xList* ⟶ PLUS | PLUS COMMA *xList*
- One or more PLUSes, each PLUS *terminated* by a semi-colon:
    1. *xList* ⟶ PLUS SEMICOLON | *xList xList*
    2. *xList* ⟶ PLUS SEMICOLON | *xList* PLUS SEMICOLON
    3. *xList* ⟶ PLUS SEMICOLON | PLUS SEMICOLON *xList*
- Zero or more PLUSes (without any separator or terminator):
    1. *xList* ⟶ ε | PLUS | *xList xList*
    2. *xList* ⟶ ε | PLUS | *xList* PLUS
    3. *xList* ⟶ ε | PLUS | PLUS *xList*
- Zero or more PLUSes, each PLUS *terminated* by a semi-colon:
    1. *xList* ⟶ ε | PLUS SEMICOLON | *xList xList*
    2. *xList* ⟶ ε | PLUS SEMICOLON | *xList* PLUS SEMICOLON
    3. *xList* ⟶ ε | PLUS SEMICOLON | PLUS SEMICOLON *xList*
- The trickiest kind of list is a list of zero or more x's, separated by commas. To get it right, think of the definition as follows:

  Either an empty list, or a non-empty list of x's separated by commas.

  We already know how to write a grammar for a non-empty list of x's separated by commas, so now it's easy to write the grammar:

$$\begin{aligned} \textit{xList} &\longrightarrow \varepsilon \mid \textit{nonemptyList} \\ \textit{nonemptyList} &\longrightarrow \texttt{PLUS} \mid \texttt{PLUS COMMA } \textit{nonemptyList} \end{aligned}$$

A Grammar for a Programming Language

To write a grammar for a whole programming language, break down the problem into pieces. For example, think about a simple Java program, which consists of one or more classes:

$$\begin{aligned} \textit{program} &\longrightarrow \textit{classList} \\ \textit{classlist} &\longrightarrow \textit{class} \mid \textit{class} \quad \textit{classList} \end{aligned}$$

A class is the word "class", optionally preceded by the word "public", followed by an identifier, followed by an open curly brace, followed by the class body, followed by a closing curly brace:

$$
\begin{array}{rcl}
\textit{class} & \longrightarrow & \texttt{PUBLIC CLASS ID LCURLY } \textit{classbody} \texttt{ RCURLY} \\
& | & \texttt{CLASS ID LCURLY } \textit{classbody} \texttt{ RCURLY}
\end{array}
$$

A class body is a list of zero or more field and/or method definitions:

$$
\begin{array}{rcl}
\textit{classbody} & \longrightarrow & \varepsilon \\
& | & \textit{deflist} \\
\textit{deflist} & \longrightarrow & \textit{def} \\
& | & \textit{def}\ \ \textit{deflist}
\end{array}
$$

and so on.

Summary

To understand how a parser works, we start by understanding **context-free grammars**, which are used to define the language recognized by the parser. Important terminology includes:

- terminal symbol
- nonterminal symbol
- grammar rule (or production)
- derivation (leftmost derivation, rightmost derivation)
- parse (or derivation) tree
- the language defined by a grammar
- ambiguous grammar

Two common kinds of grammars are grammars for **expression** languages, and grammar for **lists**. It is important to know how to write a grammar for an expression language that expresses operator precedence and associativity. It is also important to know how to write grammars for both non-empty and possibly empty lists, and for lists both with and without separators and terminators.

Contents

- Motivation and Definition
- Example 1: Value of an Arithmetic Expression
- Example 2: Type of an Expression
- Test Yourself #1
- Building an Abstract-Syntax Tree
  - The AST vs the Parse Tree
  - Translation Rules That Build an AST
  - Test Yourself #2

Motivation and Definition

Recall that the parser must produce output (e.g., an abstract-syntax tree) for the next phase of the compiler. This involves doing a **syntax-directed translation** -- translating from a sequence of tokens to some other form, based on the underlying syntax.

A syntax-directed translation is defined by augmenting the CFG: a **translation rule** is defined for each production. A translation rule defines the translation of the left-hand side nonterminal as a function of:

- constants
- the right-hand-side nonterminals' translations
- the right-hand-side tokens' values (e.g., the integer value associated with an `INTLITERAL` token, or the String value associated with an ID token)

To translate an input string:

1. Build the parse tree.
2. Use the translation rules to compute the translation of each nonterminal in the tree, working bottom up (since a nonterminal's value may depend on the value of the symbols on the right-hand side, you need to work bottom-up so that those values are available).

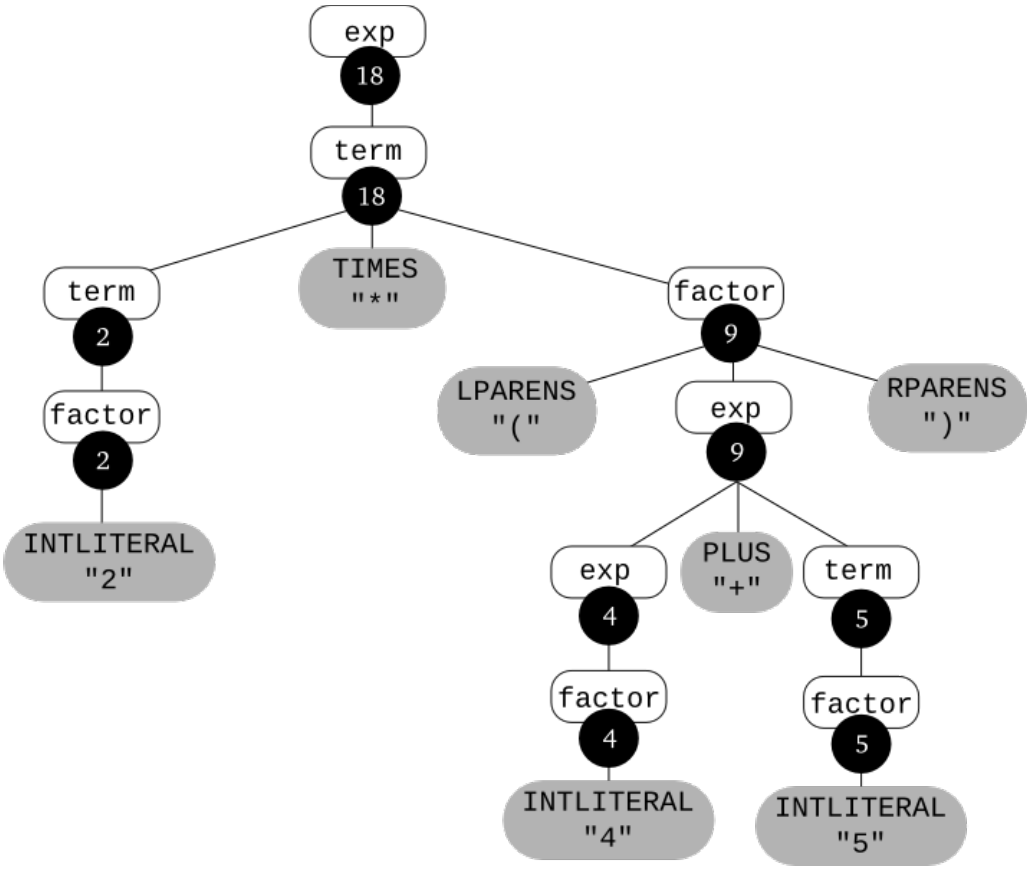
The translation of the string is the translation of the parse tree's root nonterminal.

Example 1

Below is the definition of a syntax-directed translation that translates an arithmetic expression to its integer value. When a nonterminal occurs more than once in a grammar rule, the corresponding translation rule uses subscripts to identify a particular instance of that nonterminal. For example, the rule *exp* → *exp* `PLUS` *term* has two *exp* nonterminals; exp_1 means the left-hand-side *exp*, and exp_2 means the right-hand-side *exp*. Also, the notation `xxx.value` is used to mean the value associated with token `xxx`.

CFG Production	Translation rules
exp → *exp* `PLUS` *term*	exp_1.trans = exp_2.trans + *term*.trans
exp → *term*	*exp*.trans = *term*.trans
term → *term* `TIMES` *factor*	$term_1$.trans = $term_2$.trans * *factor*.trans
term → *factor*	*term*.trans = *factor*.trans
factor → `INTLITERAL`	*factor*.trans = `INTLITERAL`.value
factor → `LPARENS` *exp* `RPARENS`	*factor*.trans = *exp*.trans

consider evaluating these rules on the input `2 * (4 + 5)`.
The result is the following annotated parse tree:



Example 2

Consider a language of expressions that use the three operators: +, &&, == using the terminal symbols `PLUS`, `AND` , `EQUALS`, respectively. Integer literals are represented by the same `INTLITERAL` token we've used before, and `TRUE` and `FALSE` represent the literals true and false (note that we could have just as well defined a single `BOOLLITERAL` token that the scanner would populate with either true or false).

Let's define a syntax-directed translation that type checks these expressions; i.e., for type-correct expressions, the translation will be the type of the expression (either **int** or **bool**), and for expressions that involve type errors, the translation will be the special value **error**. We'll use the following type rules:

1. Both operands of the + operator must be of type **int**.
2. Both operands of the && operator must be of type **bool**.
3. Both operands of the == operator must have the same (non-**error**) type.

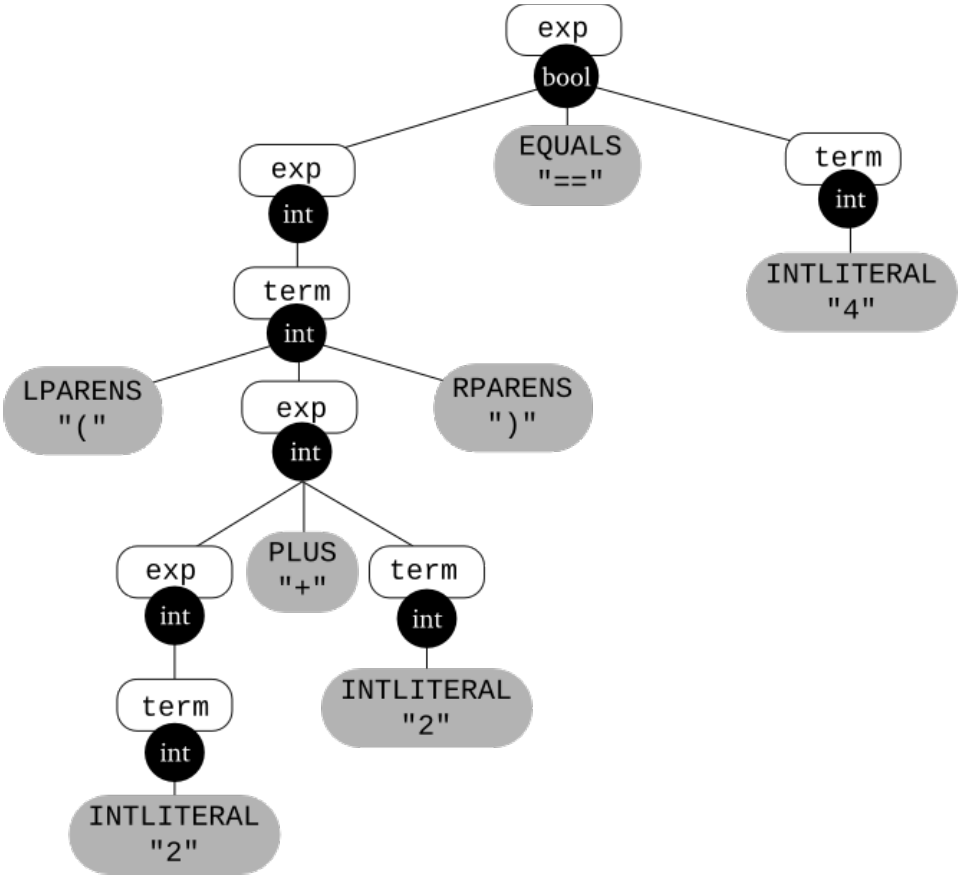
Here is the CFG and the translation rules:

CFG Production		**Translation rules**

exp	*exp* `PLUS` *term*	if (*exp*$_2$.trans == **int** and (*term*.trans == **int**) then
		exp$_1$.trans = **int**
		else
		exp$_1$.trans = **error**

exp	⟶	*exp* `AND` *term*	if (*exp*$_2$.trans == **bool** and (*term*.trans == **bool**) then *exp*$_1$.trans = **bool** else *exp*$_1$.trans = **error**
exp	⟶	*exp* `EQUALS` *term*	if (*exp*$_2$.trans == *term*.trans) and (*term*.trans ≠ **error**) then *exp*$_1$.trans = **bool** else *exp*$_1$.trans = **error**
exp	⟶	*term*	*exp*.trans = *term*.trans
term	⟶	`TRUE`	*term*.trans = **bool**
term	⟶	`FALSE`	*term*.trans = **bool**
term	⟶	`INTLITERAL`	*term*.trans = **int**
term	⟶	`LPARENS` *exp* `RPARENS`	*term*.trans = *exp*.trans

Here's an annotated parse tree for the input `(2 + 2) == 4`



TEST YOURSELF #1

The following grammar defines the language of base-2 numbers:

```
B -> 0
  -> 1
  -> B 0
  -> B 1
```

Define a syntax-directed translation so that the translation of a binary number is its base 10 value. Illustrate your translation scheme by drawing the parse tree for `1001` and annotating each nonterminal in the tree with its translation.

solution

Building an Abstract-Syntax Tree

So far, our example syntax-directed translations have produced simple values (an int or a type) as the translation of an input. In practice however, we want the parser to build an abstract-syntax tree as the translation of an input program. But that is not really so different from what we've seen so far; we just need to use tree-building operations in the translation rules instead of, e.g., arithmetic operations.

The AST vs the Parse Tree

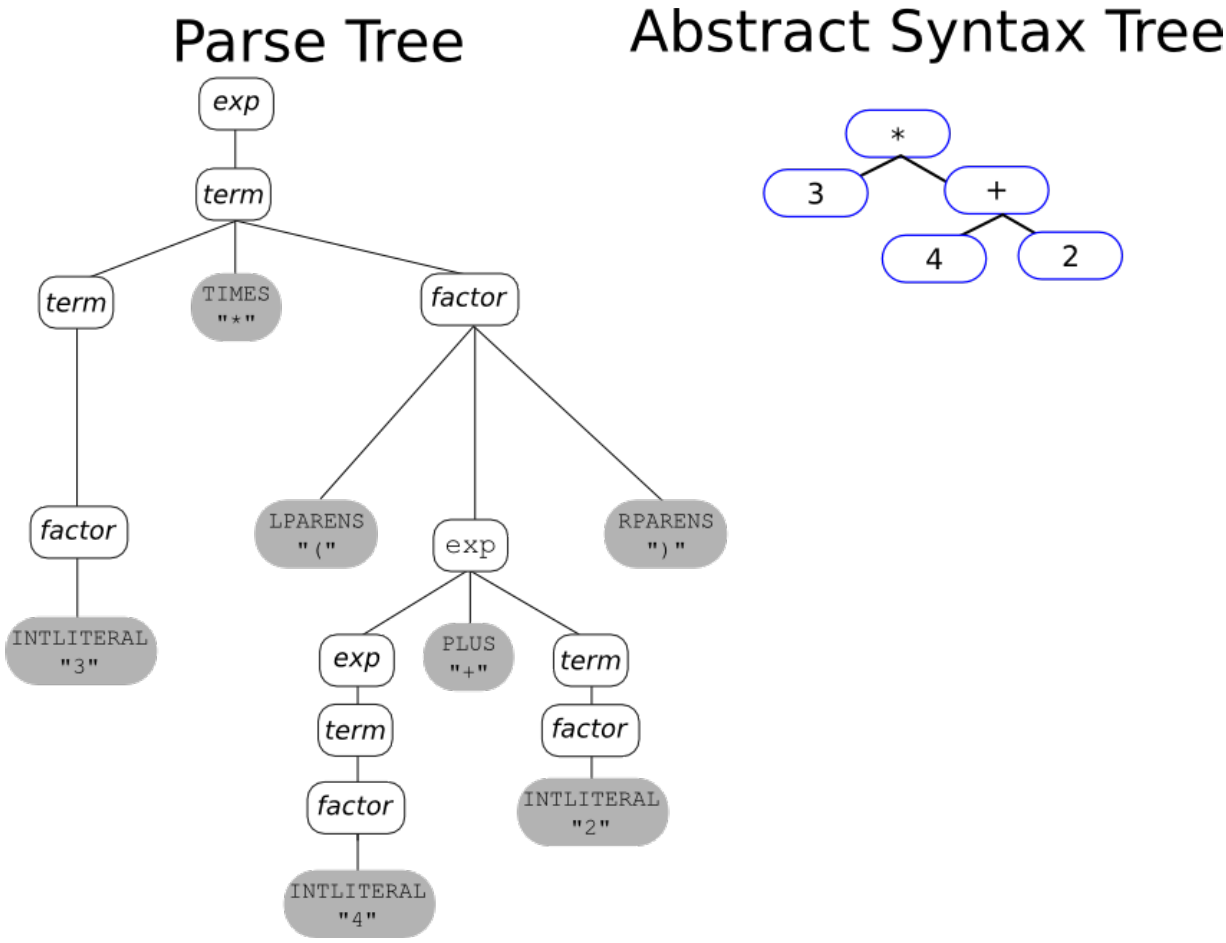
First, let's consider how an abstract-syntax tree (AST) differs from a parse tree. An AST can be thought of as a *condensed* form of the parse tree:

- Operators appear at *internal* nodes instead of at leaves.
- "Chains" of single productions are collapsed.

- Lists are "flattened".
- Syntactic details (e.g., parentheses, commas, semi-colons) are omitted.

In general, the AST is a better structure for later stages of the compiler because it omits details having to do with the source language, and just contains information about the essential structure of the program.

Below is an example of the parse tree and the AST for the expression `3 * (4 + 2)` (using the usual arithmetic-expression grammar that reflects the precedences and associativities of the operators). Note that the parentheses are not needed in the AST because the structure of the AST defines how the subexpressions are grouped.

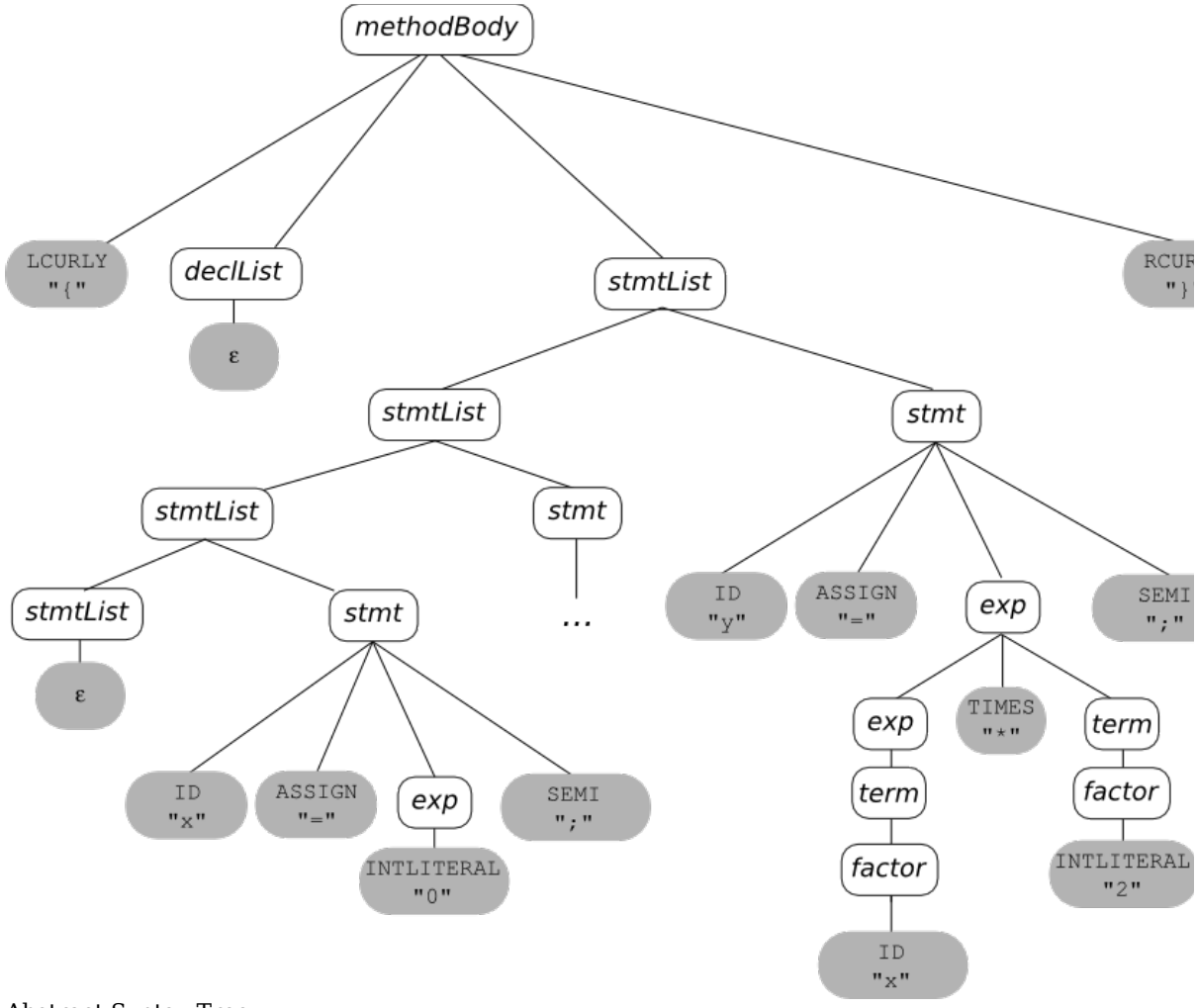


For constructs other than expressions, the compiler writer has some choices when defining the AST -- but remember that lists (e.g., lists of declarations lists of statements, lists of parameters) should be flattened, that operators (e.g., "assign", "while", "if") go at internal nodes, not at leaves, and that syntactic details are omitted.

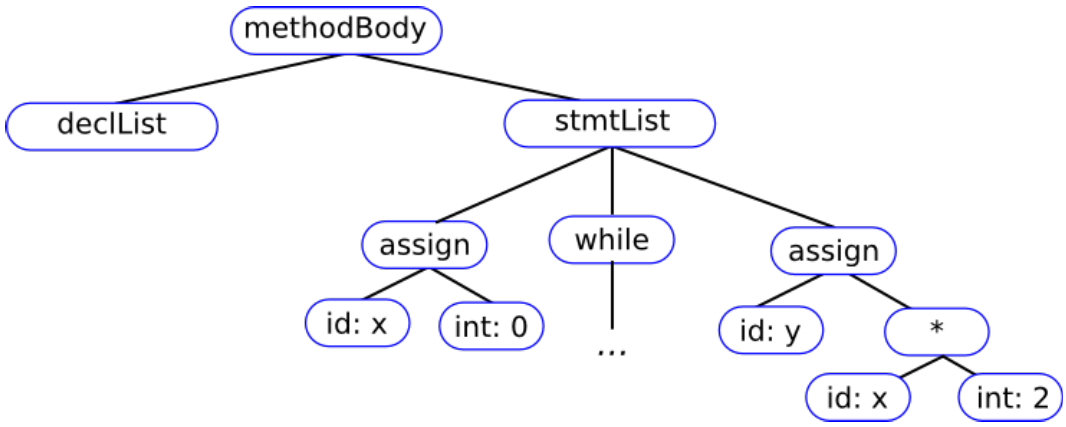
For example:

```
Input
=====

{
   x = 0;
   while (x<10) {
      x = x+1;
   }
   y = x*2;
}
```

Parse Tree:



Abstract Syntax Tree:



Note that in the AST there is just one `stmtList` node, with a list of three children (the list of statements has been "flattened"). Also, the "operators" for the statements (`assign` and `while`) have been "moved up" to internal nodes (instead of appearing as tokens at the leaves). And finally, syntactic details (curly braces and semi-colons) have been omitted.

Translation Rules That Build an AST

To define a syntax-directed translation so that the translation of an input is the corresponding AST, we first need operations that create AST nodes. Let's use java code, and assume that we have the following class hierarchy:

```
class ExpNode { }

class IntLitNode extends ExpNode {
    public IntLitNode(int val) {...}
}

class PlusNode extends ExpNode {
    public PlusNode( ExpNode e1, ExpNode e2 ) { ... }
}

class TimesNode extends ExpNode {
    public TimesNode( ExpNode e1, ExpNode e2 ) { ... }
}
```

Now we can define a syntax-directed translation for simple arithmetic expressions, so that the translation of an expression is its AST:

CFG Production			Translation rules
exp	⟶	*exp* `PLUS` *term*	exp_1.trans = new PlusNode(*exp*$_2$, *term*.trans)
exp	⟶	*term*	*exp*.trans = *term*.trans
term	⟶	*term* `TIMES` *factor*	$term_1$.trans = new TimesNode($term_2$.trans, *factor*.trans)
term	⟶	*factor*	*term*.trans = *factor*.trans
factor	⟶	`INTLITERAL`	*factor*.trans = new IntLitNode(`INTLITERAL`.value)
factor	⟶	`LPARENS` *exp* `RPARENS`	*factor*.trans = *exp*.trans

TEST YOURSELF #2

Illustrate the syntax-directed translation defined above by drawing the parse tree for the expression `2 + 3 * 4`, and annotating the parse tree with its translation (i.e., each nonterminal in the tree will have a pointer to the AST node that is the root of the subtree of the AST that is the nonterminal's translation).

solution

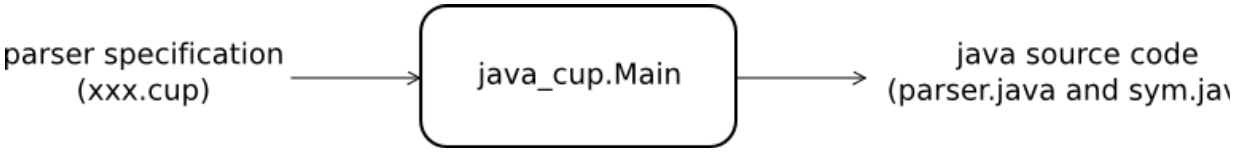
Contents

- Overview
- User Code
- Terminal and Nonterminal Declarations
- Precedence Declarations
- Grammar Rules
- How to Run Java Cup

Overview

There is a link to the Java Cup User's Manual under "Useful Programming Tools" on the class web page. Here is the same link.

Java Cup is a parser generator that produces a parser written in Java. Here's a picture illustrating how to create a parser using Java Cup:



The input to Java Cup is a specification that includes:

- optional package and import declarations
- optional user code
- terminal and nonterminal declarations
- optional precedence and associativity declarations
- grammar rules with associated actions

The key part of the specification is the last part: the grammar rules with associated actions. Those actions are like the *syntax-directed translations* rules that we have studied; i.e., they define how to translate an input sequence of tokens into some value (e.g., an abstract-syntax tree).

The output of Java Cup includes a Java source file named **parser.java**, which defines a class named **parser** with a method named **parse**. Java Cup also produces a Java source file named **sym.java**, which contains a class named **sym** that declares one public final static int for each terminal declared in the Java Cup specification.

The **parser** class has a one-argument constructor; the argument is of type **Yylex** (i.e., a scanner). The **parse** method of the **parser** class uses the given scanner to translate the input (the input stream is an argument passed to the scanner's constructor) to a sequence of tokens. It parses the tokens according to the given grammar, and does a syntax-directed translation of the input using the actions associated with the grammar productions. If the input is not syntactically correct, the parser gives an error message and quits (i.e., it only finds the first syntax error); otherwise, it returns a **Symbol** whose **value** field contains the translation of the root nonterminal (as defined by the actions associated with the grammar rules).

User Code

See the Java Cup Reference Manual for a description of this part of the specification.

Terminal and Nonterminal Declarations

All terminal and nonterminal symbols that appear in the grammar must be declared. If you want to make use of the value associated with a terminal (the **value** field of the `Symbol` object returned by the scanner for that token) in your syntax-directed translation, then you must also declare the type of that value field. Similarly, you must declare the types of the translations associated with all of the nonterminals.

```
terminal          name1, name2, ... ;  /* terminals without values */
terminal     type name1, name2, ... ;  /* terminals with values */
non terminal type name1, name2, ... ;  /* nonterminals */
```

Note that Java Cup has some reserved words (e.g., action, parser, import); these cannot be used as terminal or nonterminal names.

Precedence Declarations

A grammar like:

```
exp -> exp PLUS exp  |  exp MINUS exp  |  exp TIMES exp  | exp EQUALS exp  |  ...
```

is **ambiguous**, and will cause conflicts: the parser will not always know how to parse an input. One way to fix the problem is to rewrite the grammar by adding new nonterminals; however, this can make the grammar less clear (and the parser less efficient). Another option is to include **precedence declarations** that specify the relative precedences of the operators, as well as their associativities.

For example:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence nonassoc EQUALS;
```

The order of precedence is low to high (i.e., in this example, PLUS and MINUS are given the lowest precedence, then TIMES and DIVIDE, then EQUALS). The **left**, **right**, and **nonassoc** declarations specify the associativity of the operators. Declaring an operator **nonassoc** means that it is not legal to have two consecutive occurrences of such operators with the same precedence (so for example, given the above declarations, the expression: `a == b == c` would cause a syntax error).

Sometimes the same operator is used as both a unary and a binary operator, and the two uses have *different* precedence levels (for example, binary minus usually has a low precedence, while unary minus has a high precedence). This case can be handled either by rewriting the grammar, or by declaring a "phony" terminal symbol (e.g., UMINUS), giving it the

appropriate precedence, and using it in the grammar rules part of the specification to specify the precedence of the operator in a particular rule (see below).

Grammar Rules

The heart of the Java Cup specification is the set of grammar rules. First, there is an optional declaration of the start nonterminal; e.g.:

```
start with program;
```

If no such declaration is included, the left-hand-side nonterminal of the first grammar rule is assumed to be the start nonterminal.

Below are three example grammar rules, preceded by the appropriate terminal and nonterminal declarations. Note that `IdTokenVal` is a type that was defined in the scanner specification; `VarDeclNode`, `TypeNode`, and `IdNode` are all subclasses of an `ASTnode` class, all defined in some other file; and `IntNode` and `BoolNode` are subclasses of `TypeNode` (defined in that same file).

```
terminal                SEMICOLON;
terminal                INT;
terminal IdTokenVal     ID;

non terminal VarDeclNode      varDecl;
non terminal TypeNode         type;
non terminal IdNode           id;

varDecl ::= type:t id:i SEMICOLON
            {: RESULT = new VarDeclNode(t, i);
            :}
            ;

type        ::= INT
            {: RESULT = new IntNode();
            :}
            ;

id          ::= ID:i
            {: RESULT = new IdNode(i.idVal);
            :}
            ;
```

In these rules, lower-case names are used for nonterminals, and upper-case names are used for terminals. The symbol "::=" is used instead of an arrow to separate the left and right-hand sides of the grammar rule. Each grammar rule ends with a semicolon.

The symbols "{:" and ":}" are used to delimit the **action** associated with the rule. An action can contain arbitrary Java code (including declarations and uses of local variables). If the left-hand-side nonterminal has been declared to have a type, the action must include an assignment to the special variable **RESULT**; this assignment sets the value of the nonterminal (its translation).

To use the translations of the right-hand-side nonterminals, and the values of the right-hand-side tokens, those symbols are followed with a colon and a name. For example, using `type:t` makes `t` the name of

the translation of nonterminal `type`, and using `ID:i` makes `i` the name of the `value` field of the `Symbol` returned by the scanner for the `ID` token.

Precedence Declarations for Grammar Rules

As discussed above, sometimes an operator needs different precedences depending on whether it is being used as a unary or a binary operator. For example, the precedence declarations given above gave MINUS the lowest precedence. This is correct for binary minus, but not for unary minus (which should have the highest precedence). To handle this, a new terminal (e.g., UMINUS) can be declared, and given the highest precedence. Then the grammar rule that uses MINUS as a unary operator can be declared to have the (high) precedence of UMINUS:

```
exp     ::= MINUS exp
        {: RESULT = ...
        :}
        %prec UMINUS
        ;
```

How to Run Java Cup

To run the parser generator, type:

```
java java_cup.Main < xxx.cup
```

where `xxx.cup` is the name of the parser specification (it can have any name, but using the `.cup` extension helps to make it clear that it is a Java Cup specification). If the specification is processed without errors, two Java source files, `parser.java` and `sym.java` will be produced.

Contents

- LL(1) Grammars and Predictive Parsers
- Test Yourself #1
- Grammar Transformations
  - Left Recursion
  - Left Factoring
  - Test Yourself #2
- FIRST and FOLLOW Sets
  - FIRST
  - FOLLOW
- Test Yourself #3
- How to Build Parse Tables
  - Test Yourself #4
- How to Code a Predictive Parser
- Test Yourself #5

LL(1) Grammars and Predictive Parsers

LL(1) grammars are parsed by top-down parsers. They construct the derivation tree starting with the start nonterminal and working down. One kind of parser for LL(1) grammars is the *predictive parser*. The idea is as follows:

- "Build" the parse tree top down (don't actually build it, just discover what it would be).
- Keep track of "work to be done" using a stack of terminals and nonterminals; the scanned tokens together with the stack contents correspond to the leaves of the current (incomplete) parse tree.
- Also use a *parse table* (or selector table) to decide how to do the parse. The rows of the table are indexed by the nonterminals of the grammar, and the columns are indexed by the tokens (including the special EOF token). Each element of the table for the row indexed by nonterminal X is either empty or contains the right-hand side of a grammar rule for X.

Here's how the predictive parser works:

- Start by pushing the special "EOF" terminal onto the stack, then push the start nonterminal and call the scanner to get the first token *t*.
- Repeat:
  - If the top-of-stack symbol is a *nonterminal* X:
    - Use nonterminal X and the current token t to index into the parse table to choose a production with X on the left-hand side (the one whose right-hand side is in Table[X][t]).
    - Pop the X from the stack and push the chosen production's right-hand side (push the symbols one at a time, from right to left).
  - If the top-of-stack symbol is a *terminal*, match it with the current token. If it matches, pop it and call the scanner to get the next token.
- until one of the following happens:
  - Top-of-stack is a nonterminal, and the parse table entry is empty: reject the input.
  - Top-of-stack is a terminal but doesn't match the current token: reject the input.
  - Stack is empty: accept the input.

Here's a very simple example, using a grammar that defines the language of balanced parentheses or square brackets, and running the parser on the input "([ ]) EOF". Note that in the examples on this page we will use sometimes name terminal symbols using single characters (such as: **(**, **)**, **[**, and **]**) instead of the token names (**lparen**, **rparen**, etc). Also note that in the picture, the top of stack is to the left.

Grammar:

S ⟶ ε | **(** S **)** | **[** S **]**

Parse Table:

	(	**)**	**[**	**]**	**EOF**
S	**(** S **)**	ε	**[** S **]**	ε	ε

Input seen so far	stack	Action
(	S **EOF**	pop, push **(** S **)**
(	**(** S **)** **EOF**	pop, scan
([	S **)** **EOF**	pop, push **[** S **]**
([	**[** S **]** **)** **EOF**	pop, scan
([]	S **]** **)** **EOF**	pop, push nothing
([]	**]** **)** **EOF**	pop, scan
([])	**)** **EOF**	pop, scan
([]) **EOF**	**EOF**	pop, scan
([]) **EOF**		empty stack: input accepted

Remember, it is not always possible to build a predictive parser given a CFG; only if the CFG is LL(1). For example, the following grammar is *not* LL(1) (but it is LL(2)):

S ⟶ **(** S **)** | **[** S **]** | **(** **)** | **[** **]**

If we try to parse an input that starts with a left paren, we are in trouble! We don't know whether to choose the first production: S ⟶ **(** S **)**, or the third one: S ⟶ **(** **)**. If the next token is a right paren, we want to push "**()**". If the next token is a left paren, we want to push the three symbols "**(**S**)**". So here we need *two* tokens of look-ahead.

TEST YOURSELF #1

Draw a picture like the one given above

to illustrate what the parser for the grammar:

```
S -> epsilon | ( S ) | [ S ]
```

does on the input: "[[]]".

solution

Grammar Transformations

We need to answer two important questions:

1. How to tell whether a grammar is LL(1).
2. How to build the parse (or selector) table for a predictive parser, given an LL(1) grammar.

It turns out that there is really just one answer: if we build the parse table and no element of the table contains more than one grammar rule right-hand side, then the grammar is LL(1).

Before saying how to build the table we will consider two properties that *preclude* a context-free grammar from being LL(1): **left-recursive** grammars and grammars that are not **left factored**. We will also consider some transformations that can be applied to such grammars to make them LL(1).

First, we will introduce one new definition:

A nonterminal X is useless if either:

1. You can't derive a sequence that includes X, or
2. You can't derive a string from X (where "string" means epsilon or a sequence of terminals).

Here are some examples of useless nonterminals :

For case 1:

S ⟶ A B
A ⟶ **+** | **-** | ε
B ⟶ **digit** | B **digit**
C ⟶ **.** B

For case 2:

S ⟶ X | Y
X ⟶ **( )**
Y ⟶ **(** Y Y **)**

Y just derives more and more nonterminals, so it is useless.

From now on "context-free grammar" means a grammar without useless nonterms.

Left Recursion

- A grammar G is ***recursive*** in a nonterminal X if X can derive a sequence of symbols that includes X, in one or more steps:

$$X \overset{+}{\Longrightarrow} \alpha X \beta$$

  where α and β are arbitrary sequences of symbols.
- G is ***left recursive*** in nonterminal X if X can derive a sequence of symbols that *starts with* X, in one or more steps:

$$X \overset{+}{\Longrightarrow} X\beta$$

  where β is an arbitrary sequence of symbols.
- G is ***immediately left recursive*** in nonterminal X if X can derive a sequence of symbols that starts with X in *one* step:

$$X \Longrightarrow X\beta$$

i.e., the grammar includes the production: X ⟶ Xβ.

In general, it is not a problem for a grammar to be recursive. However, if a grammar is *left* recursive, it is not LL(1). Fortunately, we can change a grammar to remove immediate left recursion without changing the language of the grammar. Here is how to do the transformation:

Given two productions of the form:

A ⟶ A α
  | β

where:

- A is a nonterminal
- α is a sequence of terminals and/or nonterminals
- β is a sequence of terminals and/or nonterminals *not* starting with A

Replace those two productions with the following three productions:

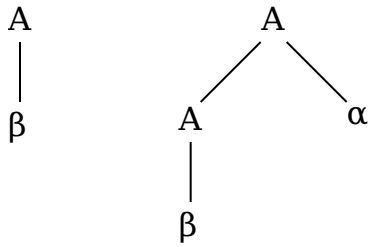
A ⟶ β A`

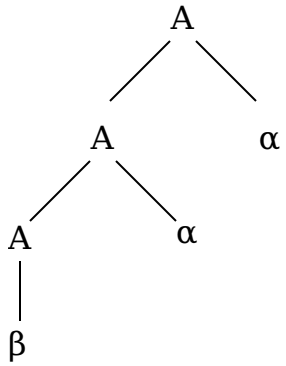
A` ⟶ α A` | ε

where A` is a new nonterminal.

Using this rule, we create a new grammar from a grammar with immediate left recursion. The new grammar is equivalent to the original one; i.e., the two grammars derive exactly the same sets of strings, but the new one is **not** immediately left recursive (and so has a chance of being LL(1)).

To illustrate why the new grammar is equivalent to the original one, consider the parse trees that can be built using the original grammar:





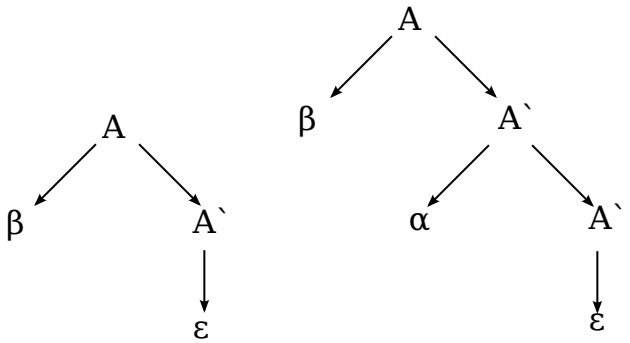
etc.

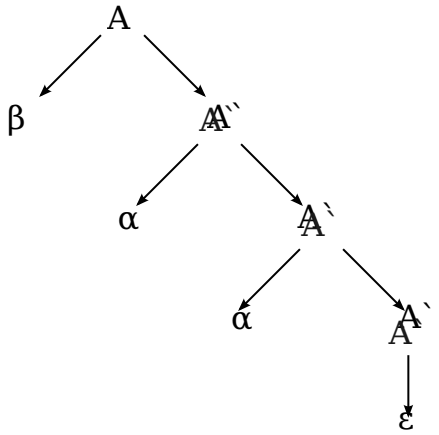
Note that the derived strings are:

- β
- β α
- β α α

  ...

That is, they are of the form "beta, followed by zero or more alphas". The new grammar derives the same set of strings, but the parse trees have a different shape (the single "beta" is derived right away, and then the zero or more alphas):





etc.

Consider, for instance, the grammar for arithmetic expressions involving only subtraction:

$$Exp \longrightarrow Exp\ \textbf{minus}\ Factor \mid Factor$$
$$Factor \longrightarrow \textbf{intliteral} \mid \textbf{(}\ Exp\ \textbf{)}$$

Notice that the first rule ($Exp \longrightarrow Exp\ \textbf{minus}\ Factor$) has immediate left recursion, so this grammar is not LL(1). (For example, if the first token is **intliteral**, you don't know whether to choose the production $Exp \longrightarrow Exp\ \textbf{minus}\ Factor$, or $Exp \longrightarrow Factor$. If the *next* token is **minus**, then you should choose $Exp \longrightarrow Exp\ \textbf{minus}\ Factor$, but if the next token is **EOF**, then you should choose $Exp \longrightarrow Factor$.

Using the transformation defined above, we remove the immediate left recursion, producing the following new grammar:

$$Exp \longrightarrow Factor\ Exp`$$
$$Exp` \longrightarrow \textbf{minus}\ Factor\ Exp` \mid$$
$$Factor \longrightarrow \textbf{intliteral} \mid \textbf{(}\ Exp\ \textbf{)}$$

Let's consider what the predictive parser built using this grammar does when the input starts with an integer:

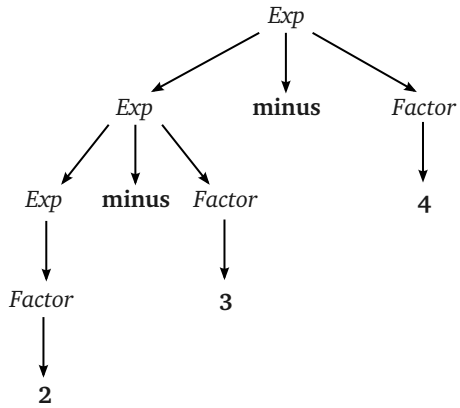
- The predictive parser starts by pushing **EOF**, then *Exp* onto the stack. Regardless of what the first token is, there is only one production with *Exp* on the left-hand side, so it will always

pop the *Exp* from the stack and push *Factor Exp`* as its first action.
- Now the top-of-stack symbol is the nonterminal *Factor*. Since the input is the **intliteral** token (not the **(** token) it will pop the *Factor* and push **intliteral**.
- The top-of-stack symbol is now a terminal (**intliteral**), which *does* match the current token, so the stack will be popped, and the scanner called to get the next token.
- Now the top-of-stack symbol is nonterminal *Exp`*. We'll consider two cases:
  1. The next token is **minus**. In this case, we pop *Exp`* and push **minus** *Factor Exp`*.
  2. The next token is **EOF**. In this case, we pop *Exp`* and push ε (i.e., push nothing).

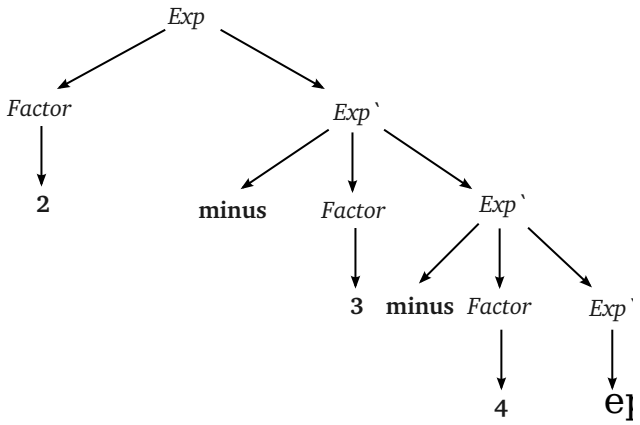
So with the new grammar, the parser is able to tell (using only one token look-ahead) what action to perform.

Unfortunately, there is a major disadvantage of the new grammar, too. Consider the parse trees for the string `2 - 3 - 4` for the old and the new grammars:

Before eliminating Left Recursion:



After eliminating Left Recursion:



The original parse tree shows the underlying structure of the expression; in particular it groups `2 - 3` in one subtree to reflect the fact that subtraction is left associative. The parse tree for the new grammar is a mess! Its subtrees don't correspond to the sub-expressions of `2 - 3 - 4` at all! Fortunately, we can design a predictive parser to create an abstract-syntax tree that *does* correctly reflect the structure of the parsed code even though the grammar's parse trees do not.

Note that the rule for removing immediate left recursion given above only handled a somewhat restricted case, where there was only one left-recursive production. Here's a more general rule for removing immediate left recursion:

- For every set of productions of the form:

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

- Replace them with the following productions:

$$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \ \ldots \ \mid \beta_n A'$$

$$A' \longrightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Note also that there are rules for removing *non-immediate* left recursion; for example, you can read about how to do that in the compiler textbook by Aho, Sethi & Ullman, on page 177. However, we will not discuss that issue here.

Left Factoring

A second property that precludes a grammar from being LL(1) is if it is not **left factored**, i.e., if a nonterminal has two productions whose right-hand sides have a common prefix. For example, the following grammar is not left factored:

Exp ⟶ **(** Exp **)** | **( )**
In this example, the common prefix is "**(**".

This problem is solved by left-factoring, as follows:

- Given a pair of productions of the form:
  A ⟶ α β_1 | α β_2
  where α is a sequence of terminals and/or nonterminals, and β_1 and $beta_2$ are sequences of terminals and/or nonterminals that do not have a common prefix (and one of the betas could be epsilon),

- Replace those two production with:
  A ⟶ α A‘
  A` ⟶ β_1 | β_2

For example, consider the following productions: Exp ⟶ **(** Exp **)** | **( )**

Using the rule defined above, they are replaced by:

Exp ⟶ **(** Exp‘
Exp‘ ⟶ Exp **)** | **)**

Here's the more general algorithm for left factoring (when there may be more than two productions with a common prefix):

For each nonterminal A:
- Find the longest non-empty prefix α that is common to 2 or more production right-hand sides.
- Replace the productions:
  A ⟶ α β_1 | … | α β_m | y_1 | .. | y_n
  with:

$$A \longrightarrow \alpha A` \mid y_1 \mid \ldots \mid y_n$$
$$A` \longrightarrow \beta_1 \mid \ldots \mid \beta_m$$

Repeat this process until no nonterminal has two productions with a common prefix.

Note that this transformation (like the one for removing immediate left recursion) has the disadvantage of making the grammar much harder to understand. However, it is necessary if you need an LL(1) grammar.

Here's an example that demonstrates both left-factoring and immediate left-recursion removal:

- The original grammar is:
  Exp ⟶ **(** Exp **)** | Exp Exp | **( )**
- After removing immediate left-recursion, the grammar becomes:
  Exp ⟶ **(** Exp **)** Exp‘ | **( )** Exp‘
  Exp‘ ⟶ Exp Exp‘ | ε
- After left-factoring, this *new* grammar becomes:
  Exp ⟶ **(** Exp‘‘
  Exp‘‘ ⟶ Exp **)** Exp‘ | **)** Exp‘
  Exp‘ ⟶ Exp Exp‘ | ε

solution

TEST YOURSELF #2

Using the same grammar: `exp -> ( exp ) | exp exp | ( )`, do left factoring first, then remove immediate left recursion.

FIRST and FOLLOW Sets

Recall: A predictive parser can only be built for an LL(1) grammar. A grammar is *not* LL(1) if it is:

◦ Left recursive, or
◦ not left factored.

However, grammars that are *not* left recursive and *are* left factored may still not be LL(1). As mentioned earlier, to see if a grammar is LL(1), we try building the parse table for the predictive parser. If any element in the table contains more than one grammar rule right-hand side, then the grammar is not LL(1).

To build the table, we must must compute **FIRST** and **FOLLOW** sets for the grammar.

FIRST Sets

Ultimately, we want to define FIRST sets for the right-hand sides of each of the grammar's productions. To do that, we define FIRST sets for arbitrary sequences of terminals and/or nonterminals, or ε (since that's what can be on the right-hand side of a grammar production). The idea is that for sequence α, FIRST(α) is the set of terminals that begin the strings derivable from α, and also, if α can derive ε, then ε is in FIRST(α). Using derivation notation:

$$\text{FIRST}(\alpha) = \{\, t \mid (t \in \Sigma \wedge \alpha \overset{*}{\Longrightarrow} t\beta) \vee (t = \varepsilon \wedge \alpha \overset{*}{\Longrightarrow} \varepsilon)\}$$

To define FIRST(α) for arbitrary α, we start by defining FIRST(X), for a *single* symbol X (a terminal, a nonterminal, or ε):

1. X ∈ Σ (e.g., X is a terminal): FIRST(X) = {X}
2. X = ε: FIRST(X) = {ε}
3. X ∈ N (e.g., X is a nonterminal): we must look at all grammar productions with X on the left, i.e., productions of the form:
   $X \longrightarrow Y_1\ Y_2\ Y_3 \ldots Y_k$
   where each Y_k is a single terminal or nonterminal (or there is just one Y,

and it is ε). For each such production, we perform the following actions:
   - Put FIRST(Y_1) - {ε} into FIRST(X).
   - If ε is in FIRST(Y_1), then put FIRST(Y_2) - {epsilon} into FIRST(X).
   - If epsilon is in FIRST(Y_2), then put FIRST(Y_3) - {ε} into FIRST(X).
   - etc...
   - If ε is in FIRST(Y_i) for $1 \leq i \leq k$ (all production right-hand sides)) then put ε into FIRST(X).
4. Repeat the previous step until there are no changes to any nonterminal's FIRST set

For example, consider computing FIRST sets for each of the nonterminals in the following grammar:

Exp ⟶ *Term Exp`*
Exp`* ⟶ **minus** *Term Exp` | ε
Term ⟶ *Factor Term'*
Term' ⟶ **divide** *Factor Term'* | ε
Factor ⟶ **intlit** | **(** *Exp* **)**

Here are the FIRST sets (starting with nonterminal factor and working up, since we need to know FIRST(factor) to compute FIRST(term), and we need to know FIRST(term) to compute FIRST(exp)):

FIRST(*Factor*) = { **intliteral**, **(**}
FIRST(*Term'*) = { **divide**, *epsilon*}

FIRST(*Term*) = { **intliteral**, **(**}
FIRST(*Exp`*) = { **minus** , *epsilon*}

FIRST(*Exp*) = { **intliteral**, **(** }

Note that FIRST(*Term*) includes FIRST(*Factor*). Since FIRST(*Factor*) does not include *epsilon*, that's all that is in FIRST(*Term*)

Note that FIRST(*Exp*) includes FIRST(*Term*). Since FIRST(*Term*) does not include *epsilon*, all that is in FIRST(*Exp*)

Once we have computed FIRST(X) for each terminal and nonterminal X, we can compute FIRST(α) for every production's right-hand-side α. In general, α will be of the form:

$X_1\ X_2\ \ldots\ X_n$

where each X is a single terminal or nonterminal, or there is just one X and it is ε. The rules for computing FIRST(α) are essentially the same as the rules for computing the first set of a nonterminal:

1. Put FIRST(X_1) - {ε} into FIRST(α)
2. If ε is in FIRST(X_1) put FIRST(X_2) into FIRST(α).
3. etc...
4. If ε is in the FIRST set for every X_k, put ε into FIRST(α).

For the example grammar above, here are the FIRST sets for each production right-hand side:

FIRST(*Term Exp'*)	=	{ **intliteral**, **(**}
FIRST(**minus** *Term Exp'*)	=	{ **minus** }
FIRST(*epsilon*)	=	{ *epsilon* }
FIRST(*Factor Term'*)	=	{ **intliteral**, **(** }
FIRST(**divide** *Factor Term'*)	=	{ **divide** }
FIRST(**intliteral**)	=	{ **intliteral** }
FIRST(**(** *Exp* **)**)	=	{ **(** }

Why do we care about the FIRST(α) sets? During parsing, suppose that there are two productions:

Production 1: $A \longrightarrow \alpha$

Production 2: $A \longrightarrow \beta$

Consider the situation when the top-of-stack symbol is *A* and the current token is **a**. If **a** ∈ FIRST(α), choose production 1: pop, push α. If, on the other hand, **a** ∈ FIRST(β), choose production 2 : pop, push β. We haven't yet given the rules for using FIRST and FOLLOW sets to determine whether a grammar is LL(1); however, you might be able to guess based on this discussion, that if **a** is in *both* FIRST(α)

and FIRST(β), the grammar is *not* LL(1).

FOLLOW Sets

FOLLOW sets are only defined for *single nonterminals*. The definition is as follows:

> For a nonterminal A, FOLLOW(A) is the set of terminals that can appear *immediately* to the right of *A* in some partial derivation; i.e., terminal **t** is in FOLLOW(*A*) if $S \overset{+}{\Longrightarrow} \ldots A\ t \ldots$ where t is a terminal. If A can be the *rightmost* symbol in a derivation, then EOF is in FOLLOW(A).

It is worth noting that ε is *never* in a FOLLOW set.

Using notation:

$$\text{FOLLOW(A)} = \{\ t \mid (t \in \Sigma \wedge S \overset{+}{\Longrightarrow} \alpha\ A\ t\ \beta) \vee (t = \text{EOF} \wedge S \overset{*}{\Longrightarrow} \alpha A)\}$$

Here are some pictures illustrating the conditions under which symbols a, c, and EOF are in the FOLLOW set of nonterminal A:

Figure 1

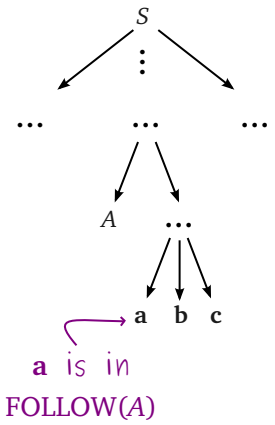


Figure 2

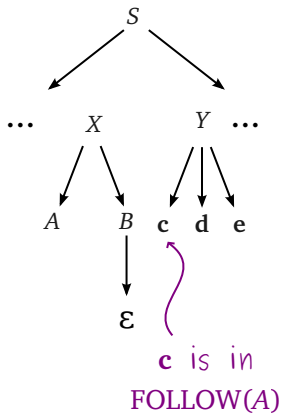
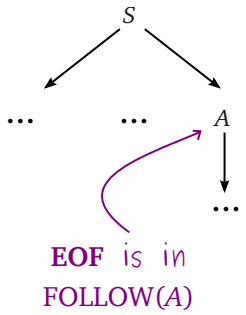


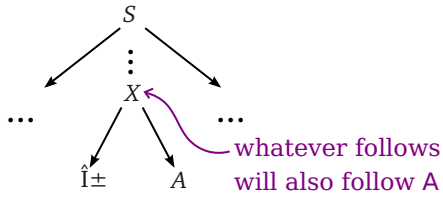
Figure 3



How to compute FOLLOW(A) for each nonterminal A:

- If A is the start nonterminal, put EOF in FOLLOW(A) (like S in Fig. 3).
- Now find the productions with A on the right-hand-side:
  - For each production X $\longrightarrow$ α A β, put FIRST(β) - {ε} in FOLLOW(A) -- see Fig. 1.
  - If epsilon is in FIRST(β) then put FOLLOW(X) into FOLLOW(A) -- see Fig. 2.
  - For each production X $\longrightarrow$ α A, put FOLLOW(X) into FOLLOW(A) -- see Figs. 3 and 4.
- Repeat the previous step until there are no changes to any nonterminal's FOLLOW set

Figure 4



It is worth noting that:

- To compute FIRST(A) you must look for A on a production's **left-hand** side.
- To compute FOLLOW(A) you must look for A on a production's **right-hand** side.
- FIRST and FOLLOW sets are always sets of **terminals** (plus, perhaps, ε for FIRST sets, and EOF for follow sets). Nonterminals are *never* in a FIRST or a FOLLOW set; ε is *never* in a FOLLOW set.

Here's an example of FOLLOW sets (and the FIRST sets we need to compute them). In this example, nonterminals are upper-case letters, and terminals are lower-case letters.

S ⟶ B **c** | D B

B ⟶ **a b** | **c** S

D ⟶ **d** | ε

α	FIRST(α)	FOLLOW(α)	
D	{**d**, ε}	{**a**, **c**}	
B	{**a**, **c**}	{**c, EOF}**	
S	{**a**, **c**, **d**}	{**EOF**, **c**}	Note that FOLLOW(S) always includes **EOF**

Now let's consider why we care about FOLLOW sets:

- Suppose, during parsing, we have some X at the top-of-stack, and **a** is the current token.
- We need to replace X on the stack with the right-hand side of a production X ⟶ α. What if X has an additional production X ⟶ β. Which one should we use?
- We've already said that if **a** is in FIRST(α), but not in FIRST(β), then we want to choose X ⟶ α.
- But what if **a** is not in FIRST(α) or FIRST(β)? If α or β can derive ε, and **a** is in FOLLOW(X), then we still have hope that the input will be accepted: If α can derive ε (i.e., ε ∈ FIRST(α), then we want to choose X ⟶ α (and similarly if β can derive ε). The idea is that since α can derive ε, it will eventually be popped from the stack, and if we're lucky, the next symbol down (the one that was under the X) will be **a**.

TEST YOURSELF #3

Here are five grammar productions for (simplified) method headers:

```
1. methodHeader -> VOID ID LPAREN paramList RPAREN

2. paramList    -> epsilon
3. paramList    -> nonEmptyParamList

4. nonEmptyParamList -> ID ID
5. nonEmptyParamList -> ID ID COMMA nonEmptyParamList
```

Question 1: Compute the FIRST and FOLLOW sets for the three nonterminals, and the FIRST sets for each production right-hand side.

Question 2: Draw a picture to illustrate what the predictive parser will do, given the input sequence of tokens: "VOID ID LPAREN RPAREN EOF". Include an explanation of how the FIRST and FOLLOW sets are used when there is a nonterminal on the top-of-stack that has more than one production.

[solution]

How to Build Parse Tables

Recall that the form of the parse table is:

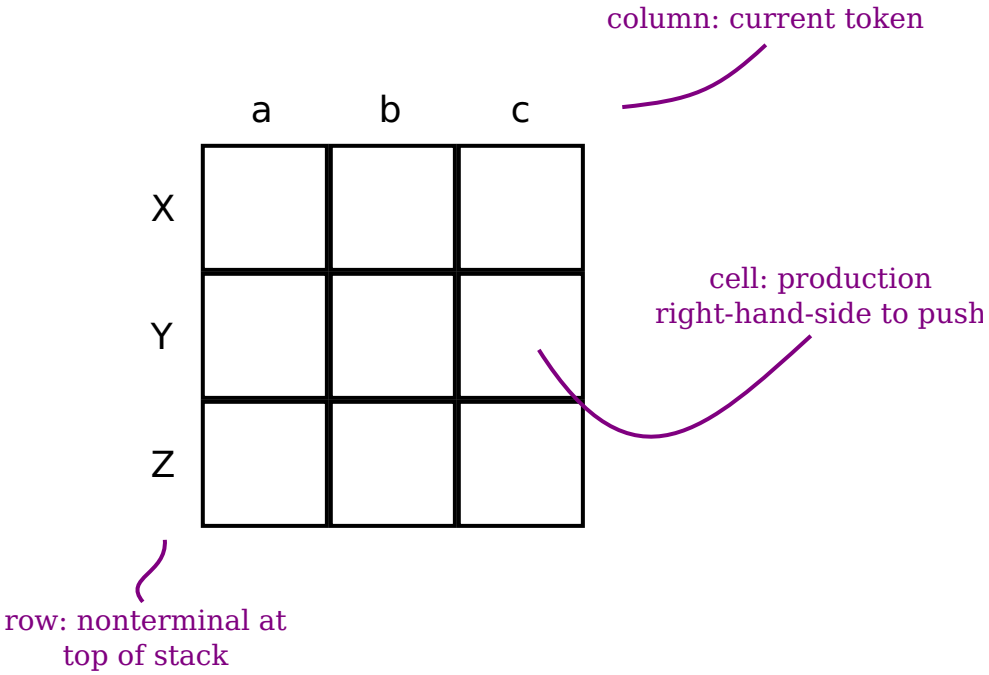


Table entry[X,**a**] is either empty (if having X on top of stack and having **a** as the current token means a syntax error) or contains the right-hand side of a production whose left-hand-side nonterminal is X -- that right-hand side is what should be pushed.

To build the table, we fill in the rows one at a time for each nonterminal X as follows:

```
for each production X ⟶ α:
   for each terminal t in FIRST(α):
      put α in Table[X,t]

   if ϵ ∈ FIRST(α) then:
      for each terminal t in FOLLOW(X):
         put α in Table[X,t]
```

The grammar is not LL(1) iff there is more than one entry for any cell in the

table.

Let's try building a parse table for the following grammar:

S ⟶ B **c** | D B

B ⟶ **a b** | **c** S

D ⟶ **d** | ε

First we calculate the FIRST and FOLLOW sets:

α	FIRST(α)	FOLLOW(α)
D	{**d**, ε}	{**a**, **c**}
B	{**a**, **c**}	{**c, EOF}**
S	{**a**, **c**, **d**}	{**EOF**, **c**}
B **c**	{**a**, **c**}	-
D B	{**d**, **a**, **c**}	-
a b	{**a**}	-
c S	{**c**}	-
d	{**d**}	-
ε	{ε}	-

Then we use those sets to start filling in the parse table:

	a	**b**	**c**	**d**	**EOF**
S	*B* **c** *D B*		*B* **c** *D B*	*D B*	
B					
D	epsilon		epsilon		

Not all entries have been filled in, but already we can see that this grammar is not LL(1) since there are two entries in table[S,a] and in table[S,c].

Here's how we filled in this much of the table:

1. First, we considered the production `S -> B c`. FIRST(`Bc`) = { a, c }, so we put the production's right-hand side (`B c`) in Table[S, a] and in Table[S, c].

FIRST(`Bc`) does not include epsilon, so we're done with that production.

2. Next, we considered the production `S -> D B` . FIRST(`DB`) = { d, a, c }, so we put the production's right-hand side (`D B`) in Table[S, d], Table[S, a], and Table[S, c].
3. Next, we considered the production `D -> epsilon`. FIRST(epsilon) = { epsilon }, so we must look at FOLLOW(D). FOLLOW(D) = { a, c }, so we put the production's right-hand side (epsilon) in Table[D, a] and Table[D, c}.

TEST YOURSELF #4

Finish filling in the parse table given above.

solution

How to Code a Predictive Parser

Now, suppose we actually want to code a predictive parser for a grammar that *is* LL(1). The simplest idea is to use a table-driven parser with an explicit stack. Here's pseudo-code for a table-driven predictive parser:

```
Stack.push(EOF);
Stack.push(start-nonterminal);
currToken = scan();

while (! Stack.empty()) {
  topOfStack = Stack.pop();
  if (isNonTerm(topOfStack)) {
    // top of stack symbol is a nonterminal
    p = table[topOfStack, currToken];
    if (p is empty) report syntax error and quit
    else {
      // p is a production's right-hand side
      push p, one symbol at a time, from right to left
    }
  }
  else {
    // top of stack symbol is a terminal
    if (topOfStack == currToken) currToken = scan();
    else report syntax error and quit
  }
}
```

TEST YOURSELF #5

Here is a CFG (with rule numbers):

```
S -> epsilon              (1)
  |  X Y Z                (2)

X -> epsilon              (3)
  |  X S                  (4)

Y -> epsilon              (5)
  |  a Y b                (6)

Z -> c Z                  (7)
  |  d                    (8)
```

Question 1(a): Compute the First and Follow Sets for each nonterminal.

Question 1(b): Draw the Parse Table.

solution

Contents

- Example: Counting Parentheses
- Test Yourself #1
- Handling Non-LL(1) Grammars
- Test Yourself #2
- Summary

Now we consider how to implement a syntax-directed translation using a predictive parser. It is not obvious how to do this, since the predictive parser works by building the parse tree top-down, while the syntax-directed translation needs to be computed bottom-up. Of course, we could design the parser to actually build the parse tree (top-down), then use the translation rules to build the translation (bottom-up). However, that would not be very efficient.

Instead, we avoid explicitly building the parse tree by giving the parser a second stack called the **semantic stack**:

- The semantic stack holds nonterminals' translations; when the parse is finished, it will hold just one value: the translation of the root nonterminal (which is the translation of the whole input).
- Values are pushed onto the semantic stack (and popped off) by adding **actions** to the grammar rules. The action for one rule must:
  - Pop the translations of all right-hand-side nonterminals.
  - Compute and push the translation of the left-hand-side nonterminal.
- The actions themselves are represented by **action numbers**, which become part of the right-hand sides of the grammar rules. They are pushed onto the (normal) stack along with the terminal and nonterminal symbols. When an action number is the top-of-stack symbol, it is popped and the action is carried out.

So what actually happens is that the action

for a grammar rule $X \longrightarrow Y_1\ Y_2\ \ldots\ Y_n$ is pushed onto the (normal) stack when the derivation step $X \longrightarrow Y_1 Y_2 \ldots Y_n$ is made, but the action is not actually performed until complete derivations for all of the Ys have been carried out.

Example: Counting Parentheses

For example, consider the following syntax-directed translation for the language of balanced parentheses and square brackets. The translation of a string in the language is the number of parenthesis pairs in the string.

CFG			**Transition Rules**
Exp	⟶	ε	Exp.trans = 0
		(Exp **)**	Exp_1.trans = Exp_2.trans + 1
	|	**[** Exp **]**	Exp_1.trans = Exp_2.trans

The first step is to replace the transition rules with **translation actions**. Each action must:

- Pop all right-hand-side nonterminals' translations from the semantic stack.
- Compute and push the left-hand-side nonterminal's translation.

Here are the transition actions:

CFG			**Transition Actions**
Exp	⟶	ε	push 0;
		(Exp **)**	`exp2trans` = pop() ; push(`exp2trans` +

		1)
	[	`exp2trans` =
|	Exp	pop() ;
	]	push(`exp2trans`)

Next, each action is represented by a unique action number, and those action numbers become part of the grammar rules:

CFG with Embedded Actions

Exp	⟶	ε #1
	|	**(** Exp **)** #2
	|	**[** Exp **]** #3

where

#1 is	push 0;
#2 is	`exp2trans` = pop() ; push(`exp2trans` + 1)
#3 is	`exp2trans` = pop() ; push(`exp2trans`)

Note that since action #3 just pushes exactly what is popped, that action is redundant, and it is not necessary to have any action associated with the third grammar rule. Here's a picture that illustrates what happens when the input "([])" is parsed (assuming that we have removed action #3):

Input so far	**Stack**	**Semantic Stack**	**Action**
(	Exp **EOF**		pop, push (exp) #2
(	**(** Exp **)** #2 **EOF**		pop, scan

([	Exp **)** #2 **EOF**		pop, push "[exp]"
([	**[** Exp **])** #2 **EOF**		pop, scan
([]	Exp **]** **)** #2 **EOF**		pop, push ε #1
([]	#1 **])** #2 **EOF**		pop, do action #1
([]	**])** #2 **EOF**	0	pop, scan
([]	**)** #2 **EOF**	0	pop, scan
([]) **EOF**	#2 **EOF**	0	pop, do action #2
([]) **EOF**	**EOF**	1	pop, scan
([]) **EOF**			empty stack, accept

translation of input = 1

In the example above, there is no grammar rule with more than one nonterminal on the right-hand side. If there were, the translation action for that rule would have to do one pop for each right-hand-side nonterminal. For example, suppose we are using a grammar that includes the rule:

MethodBody ⟶ **{** VarDecls Stmts **}**

and that the syntax-directed translation is counting the number of declarations and statements in each method body (so the translation of VarDecls is the number of derived declarations, the translation of Stmts is the number of derived statements, and the translation of MethodBody is the number of derived declarations and

statements).

```
CFG Rule:              methodBody -> { varDecls stmts }
Translation Rule:      methodBody.trans = varDecls.trans + stmts.trans
Translation Action:    stmtsTrans = pop(); declsTrans = pop();
                       push( stmtsTrans + declsTrans );
CFG rule with Action:  methodBody -> { varDecls stmts } #1
                       #1: stmtsTrans = pop();
                           declsTrans = pop();
                           push( stmtsTrans + declsTrans );
```

Note that the right-hand-side nonterminals' translations are popped from the semantic stack *right-to-left*. That is because the predictive parser does a leftmost derivation, so the `varDecls` nonterminal gets "expanded" first; i.e., its parse tree is created before the parse tree for the `stmts` nonterminal. This means that the actions that create the translation of the `varDecls` nonterminal are performed first, and thus its translation is pushed onto the semantic stack first.

Another issue that has not been illustrated yet arises when a left-hand-side nonterminal's translation depends on the value of a right-hand-side *terminal*. In that case, it is important to put the action number *before* that terminal symbol when incorporating actions into grammar rules. This is because a terminal symbol's value is available during the parse only when it is the "current token". For example, if the translation of an arithmetic expression is the value of the expression:

```
CFG Rule:              factor -> INTLITERAL
Translation Rule:      factor.trans = INTLITERAL.value
Translation Action:    push( INTLITERAL.value )
CFG rule with Action:  factor -> #1 INTLITERAL       // action BEFORE terminal
                       #1: push( currToken.value )
```

TEST YOURSELF #1

For the following grammar, give (a) translation rules, (b) translation actions with numbers, and (c) a CFG with action numbers, so that the translation of an input expression is the value of the expression. Do not worry about the fact that the

grammar is not LL(1).

```
exp    -> exp + term
       -> exp - term
       -> term
term   -> term * factor
       -> term / factor
       -> factor
factor -> INTLITERAL
       -> ( exp )
```

solution

Handling Non-LL(1) Grammars

Recall that a non-LL(1) grammar must be transformed to an equivalent LL(1) grammar if it is to be parsed using a predictive parser. Recall also that the transformed grammar usually does not reflect the underlying structure the way the original grammar did. For example, when left recursion is removed from the grammar for arithmetic expressions, we get grammar rules like this:

CFG		
Exp	⟶	Term Exp‘
Exp‘	⟶	ε
	|	**+** Term Exp‘

It is not at all clear how to define a syntax-directed translation for rules like these. The solution is to define the syntax-directed translation using the *original* grammar (define translation rules, convert them to actions that push and pop using the semantic stack, and then incorporate the action numbers into the grammar rules). Then convert the grammar to be LL(1), *treating the action numbers just like terminal grammar symbols*!

For example:

Non-LL(1) Grammar Rules With Actions

Exp	⟶	Exp + Term #1
	|	Term
Term	⟶	Term * Factor #2
	|	Factor

#1 is	`TTrans` = pop() ; `ETrans` = pop() ; push(`ETrans` + `TTrans`);
#2 is	`FTrans` = pop() ; `TTrans` = pop() ; push(`TTrans` * `FTrans`);

After Removing Immediate Left Recursion

Exp	⟶	Term Exp‘
Exp‘	⟶	+ Term #1 Exp‘
	|	ε

Term	⟶	Factor Term‘
Term‘	⟶	* Factor #2 Term‘
	|	ε

TEST YOURSELF #2

Transform the grammar rules with actions that you wrote for the "Test Yourself #1" exercise to LL(1) form. Trace the actions of the predictive parser on the input `2 + 3 * 4`.

solution

Summary

A **syntax-directed translation** is used to define the translation of a sequence of tokens to some other value, based on a CFG for the input. A syntax-directed translation is defined by associating a translation rule with each grammar rule. A translation rule defines the translation of the left-hand-side nonterminal as a function of the right-hand-side nonterminals' translations, and the values of the right-hand-side terminals. To

compute the translation of a string, build the parse tree, and use the translation rules to compute the translation of each nonterminal in the tree, bottom-up; the translation of the string is the translation of the root nonterminal.

There is no restriction on the type of a translation; it can be a simple type like an integer, or a complex type list an abstract-syntax tree.

To **implement** a syntax-directed translation using a predictive parser, the translation rules are converted to actions that manipulate the parser's **semantic stack**. Each action must pop all right-hand-side nonterminals' translations from the semantic stack, then compute and push the left-hand-side nonterminal's translation. Next, the actions are incorporated (as action numbers) into the grammar rules. Finally, the grammar is converted to LL(1) form (treating the action numbers just like terminal or nonterminal symbols).

Contents

- Introduction
- Symbol Tables
  - Scoping
  - Test Yourself #1
  - Test Yourself #2
  - Symbol Table Implementations
    - Method 1: List of Hashtables
      - Test Yourself #3
    - Method 2: Hashtable of Lists
      - Test Yourself #4
- Type Checking
  - Test Yourself #5

Introduction

The parser ensures that the input program is syntactically correct, but there are other kinds of correctness that it cannot (or usually does not) enforce. For example:

- A variable should not be declared more than once in the same scope.
- A variable should not be used before being declared.
- The type of the left-hand side of an assignment should match the type of the right-hand side.
- Methods should be called with the right number and types of arguments.

The next phase of the compiler after the parser, sometimes called the **static semantic analyzer** is in charge of checking for these kinds of errors. The checks can be done in two phases, each of which involves traversing the abstract-syntax tree created by the parser:

1. For each scope in the program: Process the declarations, adding new entries to the symbol table and reporting any variables that are multiply declared; process the statements, finding uses of undeclared variables, and updating the "ID" nodes

of the abstract-syntax tree to point to the appropriate symbol-table entry.
2. Process all of the statements in the program again, using the symbol-table information to determine the type of each expression, and finding type errors.

Below, we will consider how to build symbol tables and how to use them to find multiply-declared and undeclared variables. We will then consider type checking.

Symbol Tables

The purpose of the symbol table is to keep track of names declared in the program. This includes names of classes, fields, methods, and variables. Each symbol table entry associates a set of attributes with one name; for example:

- which kind of name it is
- what is its type
- what is its nesting level
- where will it be found at runtime.

One factor that will influence the design of the symbol table is what **scoping rules** are defined for the language being compiled. Let's consider some different kinds of scoping rules before continuing our discussion of symbol tables.

Scoping

A language's scoping rules tell you when you're allowed to reuse a name, and how to match uses of a name to the corresponding declaration. In most languages, the same name can be declared multiple times under certain circumstances. In Java you can use the same name in more than one declaration if the declarations involve different *kinds* of names. For example, you can use the same name for a class, a field of the class, a method of the class, and a local variable of the method (this is not recommended, but it is legal):

```
class Test {
  int Test;

  void Test(  ) {
      double Test;  // could also be declared int
  }
}
```

In Java (and C++), you can also use the same name for more than one method as long as the number and/or types of parameters are unique (this is called *overloading*).

In C and C++, but not in Java, you can declare variables with the same name in different blocks. A block is a piece of code inside curly braces; for example, in an **if** or a loop. The following is a legal C or C++ function, but not a legal Java method:

```
void f(int k) {
   int x = 0;       /* x is declared here */

   while (...) {
      int x = 1;   /* another x is declared here */
      ...
      if (...) {
        float x = 5.5;  /* and yet another x is declared here! */
        ...
      }
   }
}
```

As mentioned above, the scopinge rules of a language determine which declaration of a named object corresponds to each use. C, C++, and Java use what is called **static scoping**; that means that the correspondence between uses and declarations is made at compile time. C and C++ use the "most closely nested" rule to match nested declarations to their uses: a use of variable x matches the declaration in the most closely enclosing scope such that the declaration precedes the use. In C and C++, there is one, outermost scope that includes the names of the global variables (the variables that are declared outside the functions) and the names of the functions that are not part of any class. Each function has one or more scopes. Both C and C++ have one scope for the parameters and the "top-level" declarations, plus one for each

block in the function (delimited by curly braces). In addition, C++ has a scope for each *for* loop: in C++ (but not in C) you can declare variables in the for-loop header.

In the example given above, the outermost scope includes just the name "f", and function f itself has three scopes:

1. The outer scope for f includes parameter k, and the variable x that is initialized to 0.
2. The next scope is for the body of the **while** loop, and includes the variable x that is initialized to 1.
3. The innermost scope is for the body of the **if**, and includes the variable x that is initialized to 5.5.

So a use of variable x inside the loop but not inside the **if** matches the declaration in the loop (has the value 1), while a use of x outside the loop (either before or after the loop) matches the declaration at the beginning of the function (has the value 0).

TEST YOURSELF #1

Question 1: Consider the names declared in the following code. For each, determine whether it is legal according to the rules used in Java.

```
class animal {
  // methods
  void attack(int animal) {
     for (int animal=0; animal<10; animal++) {
         int attack;
     }
  }

  int attack(int x) {
     for (int attack=0; attack<10; attack++) {
        int animal;
     }
  }

  void animal() { }

  // fields
  double attack;
  int attack;
  int animal;
}
```

Question 2: Consider the following C++ code. For each use of a name, determine which declaration it corresponds to (or whether it is a use of an undeclared name).

```
int k=10, x=20;

void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
        int x;
        if (x == k) {
           int k, y;
           k = y = x;
        }
        if (x == k) {
           int x = y;
        }
    }
}
```

solution

Not all languages use static scoping. Lisp, APL, and Snobol use what is called **dynamic** scoping. A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the **most-recently-called still active** function. For example, consider the following code:

```
void main() {
  f1();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  String x = "hello";
  f3();
  g();
}

void f3() {
  double x = 30.5;
}

void g() {
  print(x);
}
```

Under dynamic scoping this program

outputs "10 hello". The first call to g comes from f1, whose copy of x has value 10. The next call to g comes from f2. Although f3 is called by f2 before it calls g, the call to f3 is not active when g is called; therefore, the use of x in g matches the declaration in f2, and "hello" is printed.

TEST YOURSELF #2

Assuming that dynamic scoping is used, what is output by the following program?

```
void main() {
  int x = 0;
  f1();
  g();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  int x = 20;
  f1();
  g();
}

void g() {
  print(x);
}
```

solution

It is generally agreed that dynamic scoping is a bad idea; it can make a program very difficult to understand, because a single use of a variable can correspond to many different declarations (with different types)! The languages that use dynamic scoping are all old languages; recently designed languages all use static scoping.

Another issue that is handled differently by different languages is whether names can be used before they are defined. For example, in Java, a method or field name *can* be used before the definition appears, but this is *not* true for a variable:

```
class Test {
  void f() {
```

```
    val = 0;   // field val has not yet been declared -- OK
    g();       // method g has not yet been declared  -- OK
    x = 1;     // variable x has not yet been declared -- ERROR!
    int x;
  }

  void g() {}
  int val;
}
```

In what follows, we will assume that we are dealing with a language that:

- uses static scoping
- requires that *all* names be declared before they are used
- does not allow multiple declarations of a name in the same scope (even for different kinds of names)
- *does* allow the same name to be declared in multiple nested scopes (but only once per scope)
- uses the same scope for a method's parameters and for the local variables declared at the beginning of the method

Symbol Table Implementations

In addition to the assumptions listed at the end of the previous section, we will assume that:

- The symbol table will be used to answer two questions:
    1. Given a declaration of a name, is there already a declaration of the same name in the current scope (i.e., is it multiply declared)?
    2. Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?
- The symbol table is only needed to answer those two questions (i.e., once all declarations have been processed to build the symbol table, and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry, the symbol table itself is no longer needed).

Given these assumptions, the symbol-table operations we will need are:

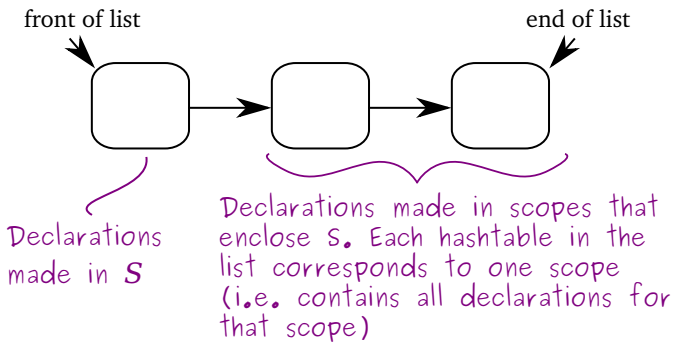
1. Look up a name in the current scope only (to check if it is multiply declared).
2. Look up a name in the current and enclosing scopes (to check for a use of an undeclared name, and to link a use with the corresponding symbol-table entry).
3. Insert a new name into the symbol table with its attributes.
4. Do what must be done when a new scope is entered.
5. Do what must be done when a scope is exited.

We will look at two ways to design a symbol table: a list of hashtables, and a hashtable of lists. For each approach, we will consider what must be done when entering and exiting a scope, when processing a declaration, and when processing a use. To keep things simple, we will assume that each symbol-table entry includes only:

- the symbol name
- its type
- the nesting level of its declaration

Method 1: List of Hashtables

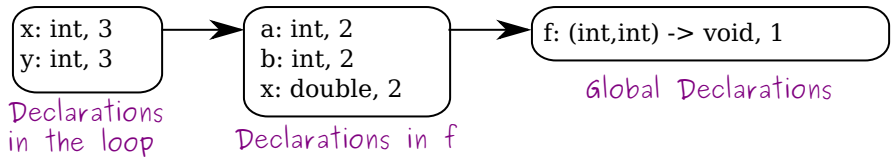
The idea behind this approach is that the symbol table consists of a list of hashtables, one for each currently visible scope. When processing a scope S, the structure of the symbol table is:



For example, given this code:

```
void f(int a, int b) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



The declaration of method g has not yet been processed, so it has no symbol-table entry yet. Note that because f is a method, its type includes the types of its parameters (int, int), and its return type (void).

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number and add a new empty hashtable to the front of the list.
2. To process a declaration of x: look up x in the first table in the list. If it is there, then issue a "multiply declared variable" error; otherwise, add x to the first table in the list.
3. To process a use of x: look up x starting in the first table in the list; if it is not there, then look up x in each successive table in the list. If it is not in *any* table then issue an "undeclared variable" error.
4. On scope exit, remove the first table from the list and decrement the current level number.

Remember that method names need to go into the hashtable for the outermost scope (not into the same table as the method's variables). For example, in the picture above, method name f is in the symbol table

for the outermost scope; name f is *not* in the same scope as parameters a and b, and variable x. This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table.

There are several factors involved in the time required for each operation:

1. **Scope entry**: time to initialize a new, empty hashtable; this is probably proportional to the size of the hashtable.
2. **Process a declaration:** using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined.
4. **Scope exit**: time to remove a table from the list, which should be O(1) if garbage collection is ignored.

TEST YOURSELF #3

For all three questions below, assume that the symbol table is implemented using a list of hashtables.

Question 1: Recall that Java does not allow the same name to be used for a local variable of a method, and for another local variable declared inside a nested scope in the method body. Even with this restriction, it is not a good idea to put *all* of a method's local variables (whether they are declared at the beginning of the method, or in some nested scope within the method body) in the *same* table. Why not?

Question 2: C++ does not use exactly the scoping rules that we have been assuming. In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name (and any uses of the name refer to the local variable).

Consider the following code. Draw the symbol table as it would be after processing the declarations in the body of f under:

- the scoping rules we have been assuming
- C++ scoping rules

```
void g(int x, int a) { }

void f(int x, int y, int z) {
  int a, b, x;
  ...
}
```

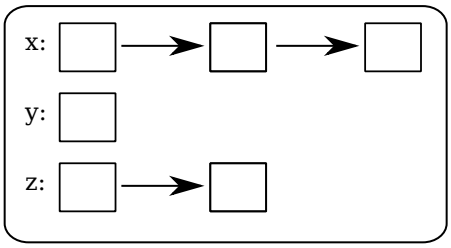
Question 3: Assume that a symbol-table entry includes the "kind" of the declared name as well as the other attributes assumed above (if the same name is declared as two different "kinds" in one scope, there would be one entry with a list of "kinds"). Also assume that we can tell (from context), for each use of a name, what "kind" of name it is supposed to be.

Which of the four operations (scope entry, process a declaration, process a use, scope exit) described above would change (and how would it change) if Java rules for name reuse were used instead of C++ rules (i.e., if the same name can be used within one scope as long as the uses are for different kinds of names, and if the same name *cannot* be used for more than one variable declaration in nested scopes)?

solution

Method 2: Hashtable of Lists

The idea behind this approach is that when processing a scope S, the structure of the symbol table is:

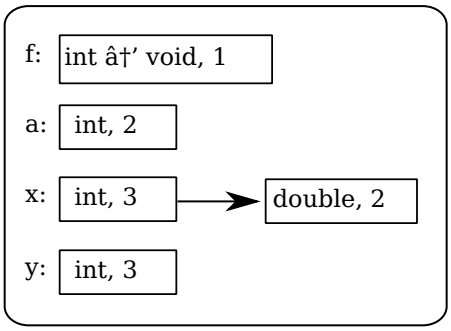


There is just one big hashtable, containing an entry for each variable for which there is some declaration in scope S or in a scope that encloses S. Associated with each variable is a list of symbol-table entries. The first list item corresponds to the most closely enclosing declaration; the other list items correspond to declarations in enclosing scopes.

For example, given this code:

```
void f(int a) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



Note that the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made in the current scope or in an enclosing scope.

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number.
2. To process a declaration of x: look up x in the symbol table. If x is there, fetch the level number from the first list item. If that level number = the current level then issue a "multiply declared variable" error; otherwise,

add a new item to the front of the list with the appropriate type and the current level number.

3. To process a use of x: look up x in the symbol table. If it is not there, then issue an "undeclared variable" error.
4. On scope exit, scan all entries in the symbol table, looking at the first item on each list. If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry). Finally, decrement the current level number.

The required times for each operation are:

1. **Scope entry**: time to increment the level number, O(1).
2. **Process a declaration**: using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing, constant expected time (O(1)).
4. **Scope exit**: time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable buckets).

TEST YOURSELF #4

Assume that the symbol table is implemented using a hashtable of lists. Draw pictures to show how the symbol table changes as the declarations in each scope in the following code is processed.

```
void g(int x, int a) {
   double d;
   while (...) {
      int d, w;
      double x, b;
      if (...) {
         int a,b,c;
      }
   }
   while (...) {
      int x,y,z;
   }
}
```

solution

Type Checking

As mentioned in the Introduction, the job of the type-checking phase is to:

- Determine the type of each expression in the program (each node in the AST that corresponds to an expression).
- Find type errors.

The **type rules** of a language define how to determine expression types, and what is considered to be an error. The type rules specify, for every operator (including assignment), what types the operands can have, and what is the type of the result. For example, both C++ and Java allow the addition of an int and a double, and the result is of type double. However, while C++ also allows a value of type double to be assigned to a variable of type int, Java considers that an error.

TEST YOURSELF #5

List as many of the operators that can be used in a Java program as you can think of (don't forget to think about the logical and relational operators as well as the arithmetic ones). For each operator, say what types the operands may have, and what is the type of the result.

In addition to finding type errors caused by operators being applied to operands of the wrong type, the type checker must also find type errors having to do with expressions that, because of their **context** must be boolean, and type errors having to do with method calls. Examples of the first kind of error include:

- the condition of an *if* statement
- the condition of a *while* loop
- the termination condition part of a *for* loop

and examples of the second kind of error include:

- calling something that is not a method
- calling a method with the wrong number of arguments
- calling a method with arguments of the wrong types

Contents

- Introduction
- Storage Layout
- Static Allocation
  - Test Yourself #1
- Stack Allocation
  - Example
  - Test Yourself #2

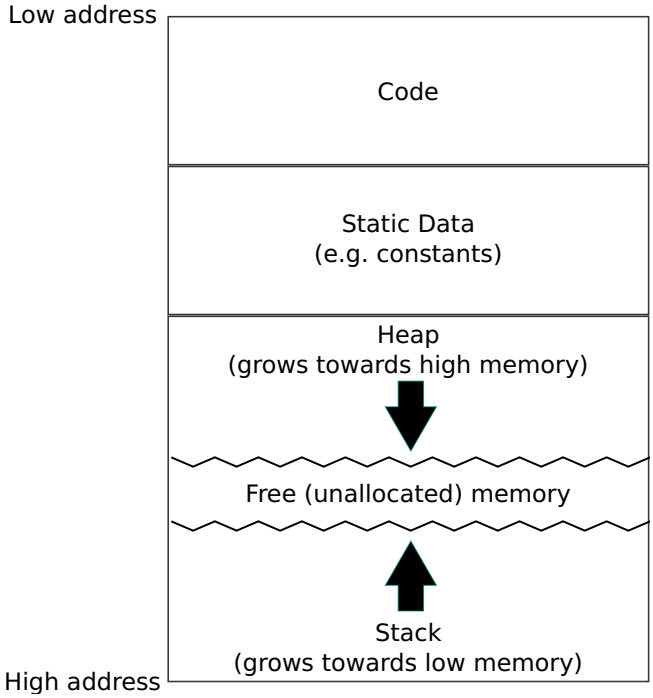
Introduction

In this set of notes we will consider:

- How storage is laid out at runtime.
- What information is stored for each method.
- What happens during method call and return for two different approaches to storage layout: static and stack allocation.

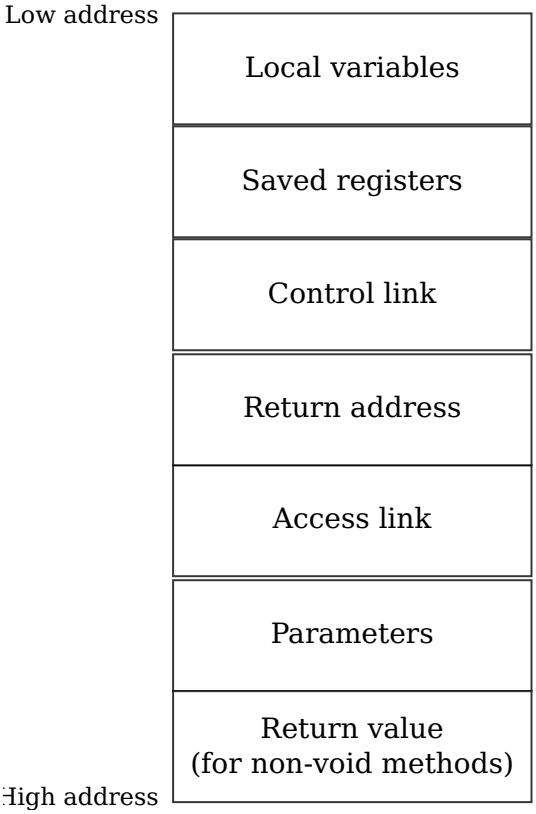
Storage Layout

There are many possible ways to organize memory. We will concentrate mainly on the standard Unix approach, illustrated below:



Usually, the **stack** is used to store one

activation record for each currently active method, and the **heap** is used for dynamically allocated memory (i.e., memory allocated as a result of using the *new* operator). An activation record is a data structure used to hold information relevant to one method call. The exact structure of an activation record depends both on the language in use and on the particular implementation; a typical organization is shown in the following picture (the individual fields will be discussed in some detail below and in the next set of notes).



As mentioned above, activation records are usually stored on the stack. A new record is pushed onto the stack when a method is called, and is popped when the method returns. However, for some languages, activation records may be stored in the heap (this might be done, for example, in a concurrent language, in which method calls do not obey the last-in-first-out protocol of a stack) or in the static data area. We will briefly consider the latter approach, then look at the most common case of stack allocation. In both cases, we will consider what must be done when a method is called, when it starts executing, and when

it returns.

Static Allocation

Some old implementations of Fortran used this approach: there is no heap or stack, and all allocation records are in the static data area, one per method. This means that every time a method is called, its parameters and local variables are stored in the same locations (which are known at compile time). This approach has some advantages and disadvantages when compared with stack or heap allocation of activation records:

ADVANTAGES

- + fast access to all names (e.g., no need to compute the address of a variable at runtime)
- + no overhead of stack/heap manipulation

DISADVANTAGES

- - no recursion
- - no dynamic allocation

Using this approach, when a method is called, the ***calling method***:

- Copies each argument into the corresponding parameter's space in the called method's activation record (AR).
- May save some registers (in its own AR).
- Performs a "Jump & Link": Jump to the first instruction of the called method, and put the address of the next instruction after the call (the return address) into the special RA register (the "return address" register).

The ***called*** method:

- Copies the return address from RA into its AR's return-address field.
- May save some registers (in its AR).
- May initialize local data.

When the called method is ready to return, it:

- Restores the values of any registers that it saved.
- Jumps to the address that it saved in its AR's return-address field.

Back in the calling method, the code that follows that call does the following:

- Restores any registers that it saved.
- If the called method was non-void (returned a value), put the return value (which may be in a special register or in the AR of the called method) in the appropriate place. For example, if the code was `x = f();`, then the return value should be copied into variable x.

TEST YOURSELF #1

Assume that static allocation is used, and that each activation record contains local variables, parameters, the return address, and (for non-void methods) the return value. Trace the execution of the following code by filling in the appropriate fields of the activation records of the three methods. Also think about where the string literals would be stored.

```
1.  void error(String name, String msg) {
2.    System.out.println("ERROR in method " + name + ": " + msg);
3.  }
4.
5.  int summation(int max) {
6.    int sum = 1;
7.    for (int k=1; k<=max; k++) {
8.      sum += k;
9.    }
10.   return sum;
11. }
12.
13. void main() {
14.   int x = summation(3);
15.   if (x != 6) error("main", "bad value returned by summation");
16. }
```

solution

Stack Allocation

Stack allocation is used to implement most modern programming languages. The basic idea is that:

- Each time a method is called, a new AR (also called a **stack frame**) is pushed onto the stack.
- The AR is popped when the method returns.
- A register (SP for "stack pointer") points to the top of the stack.
- Another register (FP for "frame pointer") points to a fixed item (such as the return address or the access link) in the current method's AR.

When a method is called, the calling method:

- May save some registers (in its own AR).
- Pushes the parameters onto the stack (into space that is shared with the called method's AR).
- If the language allows nested methods, may set up the access link; this means pushing the appropriate value -- more on that in the next set of notes -- onto the stack.
- Does a "Jump & Link" -- jumps to the 1st instruction of the called method, and puts the address of the next instruction (the one after the call) into register RA.

The called method:

- Pushes the return address (from RA) onto the stack (into its AR's "return address" field).
- Pushes the old FP into its AR's "control link" field.
- Sets the FP to point to the appropriate place in its AR (to the "access link" field if there is one; otherwise, to the "return-address" field). The address of that field is computed as follows: SP + (size of "control link" field) + (size of "return address" field) + (size of "access link"

field). All of these sizes are computed at *compile* time. (Note that values are *added* to the SP because we are assuming that "lower" on the stack means a *higher* address.)
- May save some registers (by pushing them onto the stack).
- Sets up the "local data" fields. This may involve pushing actual values if the locals are initialized as part of their declarations, or it may just involve subtracting their total size from the SP.

When the method returns, it:

- Restores the values of any saved registers.
- Loads the return address into register RA (from the AR).
- Restores the old stack pointer (SP = FP).
- Restores the old frame pointer (FP = saved FP, i.e., the value in the control-link field).
- Return (jump to the address in register RA).

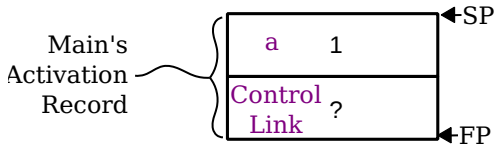
Example: Activation Records

Consider the following code:

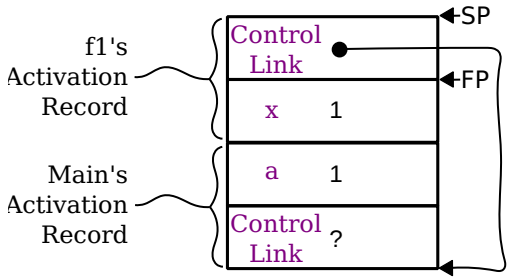
```
void f2(int y) {
  f1(y);
}

void f1(int x) {
  if (x > 0) f2(x-1);
}

main() {
  int a = 1;
  f(1);
}
```

The following pictures show the activation records on the stack at different points during the code's execution (only the control link, parameter, and local variable fields are shown).

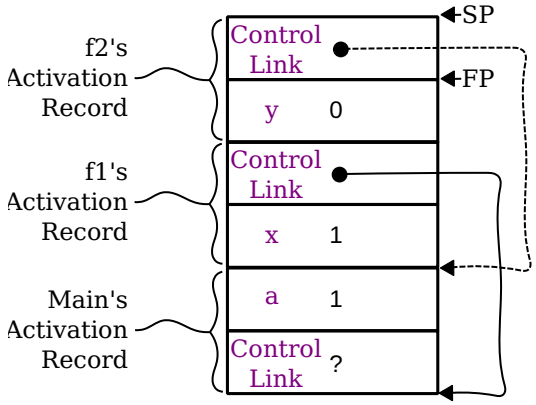
1. When the program starts:



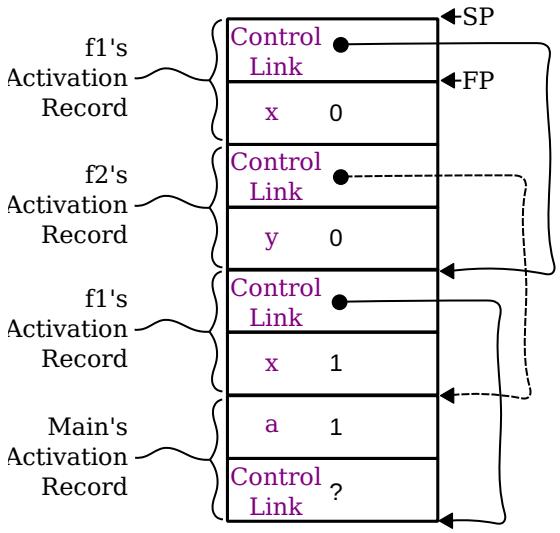
2. After main calls f1:



3. After f1 calls f2:



4. After f2 calls f1:



After this, f1 returns (and its AR is popped), then f2 returns, then the first call to f1 returns, then the whole program ends.

TEST YOURSELF #2

Assume that stack allocation is used. Trace the execution of the following code by filling in the local variables, parameters, and control link fields of the activation records (recall that dynamically allocated storage is stored in the heap, not on the stack).

```
1.  void init(int[] A, int len) {
2.    for (int k=1; k<len; k++) {
3.      A[k] = k;
4.    }
5.  }
6.
7.  void main() {
8.    int[] x = new int[3];
9.    init(x, 3);
10. }
```

solution

Contents

- Overview
- Value Parameters
  - Test Yourself #1
- Reference Parameters
  - Test Yourself #2
  - Test Yourself #3
- Value-Result Parameters
  - Creating Aliases via Pointers
  - Creating Aliases by Passing Globals as Arguments
  - Creating Aliases by Passing Same Argument Twice
  - Test Yourself #4
- Name Parameters
- Comparison

Overview

In a Java program, all parameters are passed by *value*. However, there are three other parameter-passing modes that have been used in programming languages:

1. pass by reference
2. pass by value-result (also called copy-restore)
3. pass by name

We will consider each of those modes, both from the point of view of the programmer and from the point of view of the compiler writer.

First, here's some useful terminology:

1. Given a method header, e.g.:

   ```
   void f(int a, boolean b, int c)
   ```

   we will use the terms ***parameters, formal parameters***, or just ***formals*** to refer to a, b, and c.

2. Given a method call, e.g.:

```
f(x, x==y, 6);
```

   we will use the terms ***arguments, actual parameters***, or just ***actuals*** to refer to x, x==y, and 6.

3. The term ***r-value*** refers to the value of an expression. So for example, assuming that variable x has been initialized to 2, and variable y has been initialized to 3:

expression	**r-value**
x	2
y	3
x+y	5
x==y	false

4. The term ***l-value*** refers to the location or address of an expression. For example, the l-value of a global variable is the location in the static data area where it is stored. The l-value of a local variable is the location on the stack where it is (currently) stored. Expressions like `x+y` and `x==y` have no l-value. However, it is not true that only identifiers have l-values; for example, if A is an array, the expression `A[x+y]` has both an r-value (the value stored in the x+yth element of the array), and an l-value (the address of that element).

L-values and r-values get their names from the Left and Right sides of an assignment statement. For example, the code generated for the statement `x = y` uses the l-value of x (the left-hand side of the assignment) and the r-value of y (the right-hand side of the assignment). Every expression has an r-value. An expression has an l-value iff it can be used on the left-hand side of an assignment.

Value Parameters

Parameters can only be passed by value in Java and in C. In Pascal, a parameter is passed by value unless the corresponding formal has the keyword ***var***; similarly, in C++, a parameter is passed by value unless the corresponding formal has the symbol ***&*** in front of its name. For example, in the Pascal and C++ code below, parameter x is passed by value, but not parameter y:

```
// Pascal procedure header
Procedure f(x: integer; var y: integer);

// C++ function header
void f(int x; int & y);
```

When a parameter is passed by value, the calling method copies the r-value of the argument into the called method's AR. Since the called method only has access to the copy, changing a formal parameter (in the called method) has ***no*** effect on the corresponding argument. Of course, if the argument is a pointer, then changing the "thing pointed to" ***does*** have an effect that can be "seen" in the calling procedure. For example, in Java, arrays are really pointers, so if an array is passed as an argument to a method, the called method can change the ***contents*** of the array, but not the array variable itself, as illustrated below:

```
void f( int[] A ) {
    A[0] = 10;   // change an element of parameter A
    A = null;    // change A itself (but not the corresponding actual)
}

void g() {
    int[] B = new int [3];
    B[0] = 5;
    f(B);
    //*** B is not null here, because B was passed by value
    //*** however, B[0] is now 10, because method f changed the first element
    //*** of the array pointed to by B
}
```

TEST YOURSELF #1

What is printed when the following Java program executes and why?

```
class Person {
```

```
    int age;
    String name;
}

class Test {
    static void changePerson(Person P) {
        P.age = 10;
        P = new Person();
        P.name = "Joe";
    }

    public static void main(String[] args) {
        Person P = new Person();
        P.age = 2;
        P.name = "Ann";
        changePerson(P);
        System.out.println(P.age);
        System.out.println(P.name);
    }
}
```

solution

Reference Parameters

When a parameter is passed by reference, the calling method copies the l-value of the argument into the called method's AR (i.e., it copies a ***pointer*** to the argument instead of copying the argument's value). Each time the formal is used, the pointer is followed. If the formal is used as an r-value (e.g., its value is printed, or assigned to another variable), the value is fetched from the location pointed to by the pointer. If the formal is assigned a new value, that new value is written into the location pointed to by the pointer (the new value is *not* written into the called method's AR).

If an argument passed by reference has no l-value (e.g., it is an expression like `x+y`), the compiler may consider this an error (that is what happens in Pascal, and is also done by some C++ compilers), or it may give a warning, then generate code to evaluate the expression, to store the result in some temporary location, and to copy the address of that location into the called method's AR (this is is done by some C++ compilers).

In terms of language design, it seems like a

good idea to consider this kind of situation an error. Here's an example of code in which an expression with no l-value is used as an argument that is passed by reference (the example was actually a Fortran program, but Java-like syntax is used here):

```
void mistake(int x) {  // x is a reference parameter
   x = x+1;
}

void main() {
   int a;
   mistake(1);
   a = 1;
   print(a);
}
```

When this program was compiled and executed, the output was 2! That was because the Fortran compiler stored 1 as a literal at some address and used that address for all the literal "1"s in the program. In particular, that address was passed when "mistake" was called, and was also used to fetch the value to be assigned into variable a. When "mistake" incremented its parameter, the location that held the value 1 was incremented; therefore, when the assignment to a was executed, the location no longer held a 1, and so a was initialized to 2!

To understand why reference parameters are useful, remember that, although in Java all non-primitive types are really pointers, that is not true in other languages. For example, consider the following C++ code:

```
class Person {
  public:
     String name;
     int age;
};

void birthday(Person per) {
   per.age++;
}

void main() {
   Person P;
   P.age = 0;
   birthday(P);
   print(P.age);
}
```

Note that in `main`, variable `P` is a `Person`, not a pointer to a `Person`; i.e., `main`'s activation record has space for `P.name` and `P.age`. Parameter `per` is passed by value (there is no ampersand), so when `birthday` is called from `main`, a copy of variable `P` is made (i.e., the values of its `name` and `age` fields are copied into `birthday`'s AR). It is only the copy of the `age` field that is updated by `birthday`, so when the print statement in `main` is executed, the value that is output is 0.

This motivates some reasons for using reference parameters:

1. When the job of the called method is to modify the parameter (e.g., to update the fields of a class), the parameter must be passed by reference so that the actual parameter, not just a copy, is updated.
2. When the called method will ***not*** modify the parameter, and the parameter is very large, it would be time-consuming to copy the parameter; it is better to pass the parameter by reference so that a single pointer can be passed.

TEST YOURSELF #2

Consider writing a method to sort the values in an array of integers. An operation that is used by many sorting algorithms is to swap the values in two array elements. This might be accomplished using a `swap` method:

```
static void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Assume that `A` is an array of 4 integers. Draw two pictures to illustrate what happens when the call:

```
swap(A[0], A[1]);
```

is executed, first assuming that this is Java code (all parameters are passed by value), and then assuming that this is some other language in which parameters are passed by reference. In both cases, assume that the array itself is stored in the heap (i.e., the space for `A` in the calling method's AR holds a pointer to the space allocated for the array in the heap). Your pictures should show the ARs of the calling method and method `swap`.

solution

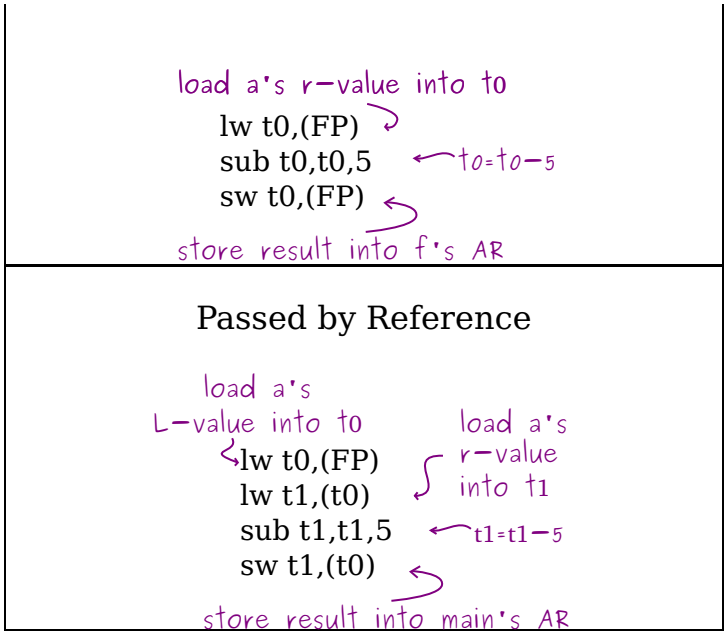
It is important to realize that the code generator will generate different code for a use of a parameter in a method, depending on whether it is passed by value or by reference. If it is passed by value, then it is in the called method's AR (accessed using an offset from the FP). However, if it is passed by reference, then it is in some other storage (another method's AR, or in the static data area). The value in the called method's AR is the address of that other location.

To make this more concrete, assume the following code:

```
void f(int a) {
   a = a - 5;
}

void main() {
    int x = 10;
    f(a);
}
```

Below is the code that would be generated for the statement `a = a - 5`, assuming (1) that `a` is passed by value and (2) assuming that `a` is passed by reference:

Passed by Value



Passed by Reference

Notice that when passing by reference the cell at address FP contains the *address* of `a` and not its value. Therefore, the first instruction will copy the location of `a` into t0 and the second instruction will extract the value at such a location and copy it into t1. Finally, the last instruction will copy the result into the location pointed by t0-i.e., the location of `a`.

TEST YOURSELF #3

The code generator will also generate different code for a method ***call*** depending on whether the arguments are to be passed by value or by reference. Consider the following code:

```
int x, y;
x = y = 3;
f(x, y);
```

solution

Assume that `f`'s first parameter is passed by reference, and that its second parameter is passed by value. What code would be generated to fill in the parameter fields of `f`'s AR?

solution

Value-Result Parameters

Value-result parameter passing was used in Fortran IV and in Ada. The idea is that, as for pass-by-value, the value (not the address) of the actual parameters are copied into the called method's AR. However, when the called method ends, the final values of the parameters are copied back into the arguments. Value-result is equivalent to call-by-reference ***except*** when there is aliasing (note: "equivalent" means the program will produce the same results, not that the same code will be generated).

Two expressions that have the same l-value are called aliases. Aliasing can happen:

- via pointer manipulation,
- when a parameter is passed by reference and is also global to the called method,
- when a parameter is passed by reference using the same expression as an argument more than once; e.g., call test(x,y,x).

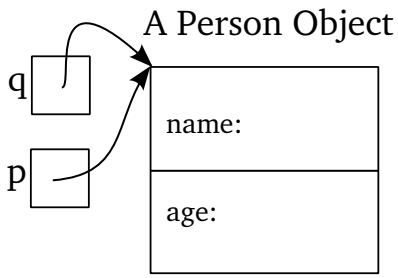
Will will look at examples of each of these below.

Creating Aliases via Pointers

Pointer manipulation can create aliases, as illustrated by the following Java code. (Note: this kind of aliasing does not make pass-by-reference different from pass-by-value-result; it is included here only for completeness of the discussion of aliasing.)

```
Person p, q;
  p = new Person();
  q = p;
  // now p.name and q.name are aliases (they both refer to the same location)
  // however, p and q are not aliases (they refer to different locations)
```

Pictorially:



Creating Aliases by Passing Globals as Arguments

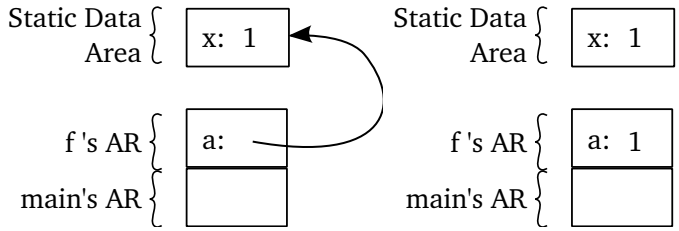
This way of creating aliases (and the difference between reference parameters and value-result parameters in the presence of this kind of aliasing) are illustrated by the following C++ code:

```
int x = 1;      // a global variable

void f(int & a)
{ a = 2;        // when f is called from main, a and x are aliases
  x = 0;
}

main()
{ f(x);
  cout << x;
}
```

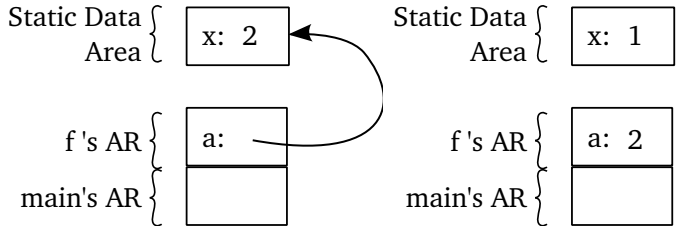
As stated above, passing parameters by value-result yields the same results as passing parameters by reference ***except*** when there is aliasing. The above code will print different values when `f`'s parameter is passed by reference than when it is passed by value-result. To understand why, look at the following pictures, which show the effect of the code on the activation records (only variables and parameters are shown in the ARs, and we assume that variable `x` is in the static data area):

Call-by-reference **Call-by-value**

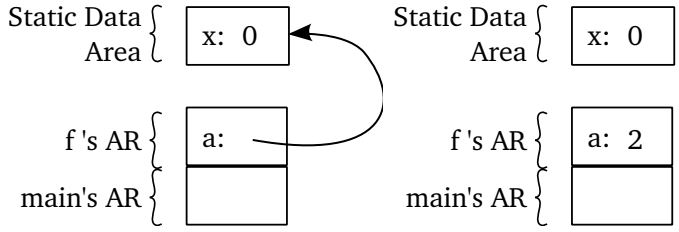
At time of call



After a = 2

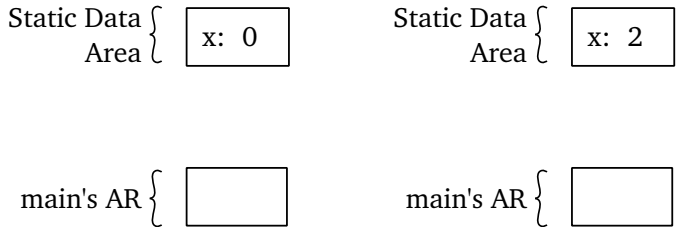


After x = 0



After call

When f returns the final value of value-result parameter `a` is copied back into the space for `x`, so:



Output

0 2

Creating Aliases by Passing Same Argument Twice

Consider the following C++ code:

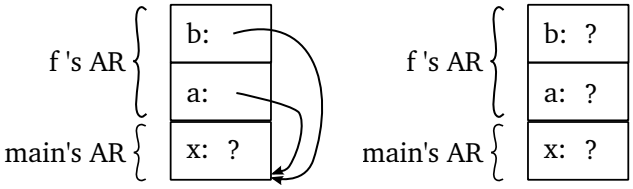
```
void f(int &a, &b)
```

```
{ a = 2;
  b = 4;
}

main()
{ int x;
  f(x, x);
  cout << x;
}
```

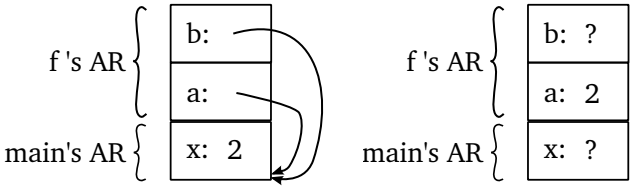
Assume that `f`'s parameters are passed by reference. In this case, when `main` calls `f`, `a` and `b` are aliases. As in the previous example, different output may be produced in this case than would be produced if `f`'s parameters were passed by value-result (in which case, no aliases would be created by the call to `f`, but there would be a question as to the order in which values were copied back after the call). Here are pictures illustrating the difference:

Call-by-reference	**Call-by-value-result**

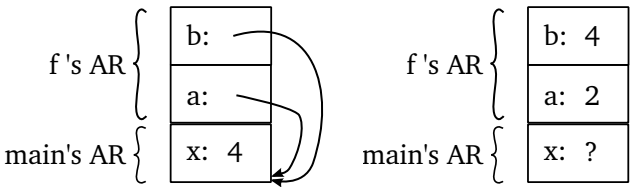
<u>At time of call</u>



<u>After a = 2</u>



<u>After b = 4</u>



After call

Output

4 ???

With value-result parameter passing, the value of `x` after the call is undefined, since it is unknown whether `a` or `b` gets copied back into `x` first. This may be handled in several ways:

- Code like this (where multiple actual parameters that are passed by value have the same l-value) may cause a compile-time error.
- The order in which parameter values are copied back after a call may be defined by the specific language.
- The order in which parameter values are copied back after a call may be left as implementation dependent (so code like the above may produce different outputs when compiled with different compilers).

TEST YOURSELF #4

Assume that all parameters are passed by value-result.

Question 1: Give a high-level description of what the code generator must do for a method call.

Question 2: Give the specific code that would be generated for the call shown below, assuming that variables x and y are stored at offsets -8 and -12 in the calling method's AR.

```
int x, y;
f(x, y);
```

solution

Name Parameters

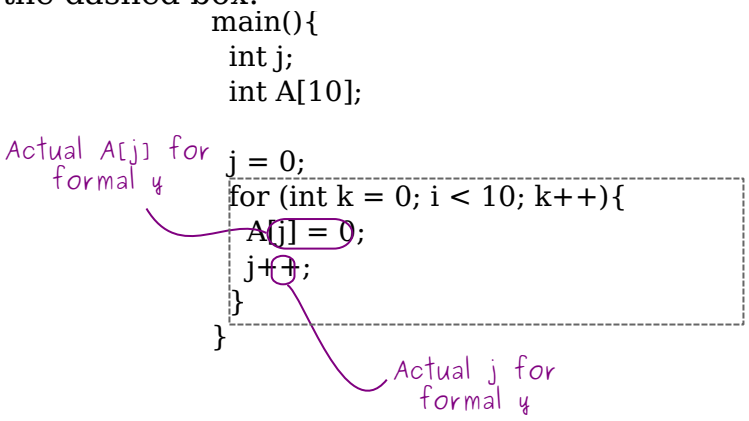
Call-by-name parameter passing was used in Algol. The way to understand it (not the way it is actually implemented) is as follows:

- Every call statement is replaced by the body of the called method.
- Each occurrence of a parameter in the called method is replaced with the corresponding argument -- the actual text of the argument, not its value.

For example, given this code:

```
void Init(int x, int y)
{ for (int k = 0; i < 10; k++){
    y = 0;
    x++;
  }
}

main()
{ int j;
  int A[10];

  j = 0;
  Init(j, A[j]);
}
```

The following shows this (conceptual) substitution, with the substituted code in the dashed box:



Call-by-name parameter passing is not really implemented like macro expansion however; it is implemented as follows. Instead of passing values or addresses as arguments, a method (actually the address

of a method) is passed for each argument. These methods are called 'thunks'. Each 'thunk' knows how to determine the address of the corresponding argument. So for the above example:

- 'thunk' for j - return address of j
- 'thunk' for A[j] - return the address of the jth element of A, using the ***current*** value of j

Each time a parameter is used, the 'thunk' is called; then the address returned by the 'thunk' is used.

For the example above, call-by-reference would execute `A[0] = 0` ten times, while call-by-name initializes the whole array!

The effect of evaluating argument expressions in the callee as needed can have some surprising effects. For example, an argument that would otherwise cause a runtime crash (say divide-by-zero) won't cause any problems until it is actually used (if at all). Factors like these often make call-by-name programs hard to understand - it may require looking at every call of a method to figure out what that method is doing.

Call-by-name is interesting for historical, research, and academic reasons, However, it is considered too confusing for developers in practice and industry has largely passed it by in favor of call-by-value or call-by-reference.

Comparisons of These Parameter Passing Mechanisms

Here are some advantages of each of the parameter-passing mechanisms discussed above:

Call-by-Value (when not used to pass pointers)

- Doesn't cause aliasing.
- Arguments unchanged by method call, so easier to understand calling code (no need to go look at called method to see what it does to actual parameters).
- Easier for static analysis (for both programmer and compiler). For example:

```
x = 0;
f(x);           {call-by-value so x not changed}
z = x + 1;      {can replace by z = 1 when optimizing}
```

- Compared with call-by-reference, the code in the called method is faster because there is no need for indirection to access formals.

Call-by-Reference

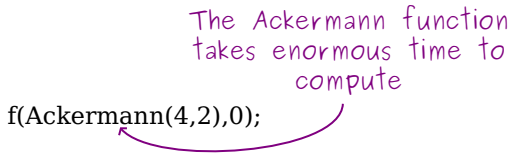
- More efficient when passing large objects (only need to copy addresses, not the objects themselves).
- Permits actuals to be modified (e.g., can implement `swap` method for integers).

Call-by-Value-Result

- As for call-by-value, more efficient than call-by-reference for small objects (because there is no overhead of pointer dereferencing for each use).
- If there is no aliasing, can implement call-by-value-result using call-by-reference for large objects, so it is still efficient.

Call-by-Name

- More efficient than other approaches when passing parameters that are never used. For example:




```
void f(int a, int b){
  if (b == 1){
    return a + 1;
  } else {
    return 0;
  }
}
```

If the condition `b` in method `f` is not 1,

then using call-by-name, it is never necessary to evaluate the first actual at all! That's good because doing so would take a ***long*** time* .

Contents

- Introduction
- Local Variables
  - Doubles
  - Test Yourself #1
- Global Variables
- Non-Local Variables
  - Static Scoping
    - Test Yourself #2
    - Method #1: Access Links
      - Test Yourself #3
      - Summary
    - Method #2: The Display
      - Test Yourself #4
    - Comparison: Access Links vs The Display
  - Dynamic Scoping
    - Deep Access
      - Test Yourself #5
    - Shallow Access
      - Test Yourself #6

Introduction

In this set of notes we will consider how three different kinds of variables: local, global, and non-local, are accessed at runtime. For the purposes of this discussion, we will define these three categories as follows:

1. **Local variables**: variables that are declared in the method that accesses them.
2. **Global variables**: variables that are not declared in any method, but are accessible in all methods. For example, the public static fields of a Java class can be used in any method; variables can be declared at the file level in a C or C++ program, and used in any function; variables can be declared in the outermost scope of a Pascal program and can be used in any procedure or function.
3. **Non-local variables**: we will use this term for two situations:
    1. In languages that allow sub-programs to be *nested*, it refers to variables that are declared in one sub-program and used in a nested sub-program. This is allowed, for example, in Pascal.
    2. In languages with dynamic scope, it refers to variables that are used in a method without being declared there (so the use corresponds to the

declaration in the "most recently called, still active" method).

Our discussion will include information specific to the last programming assignment (the code generator); i.e., how to generate MIPS code to access these three kinds of variables.

Local Variables

Local variables and parameters are stored in the activation record of the method in which they are declared. They are accessed at runtime using an offset relative to the frame pointer (FP). Since we're assuming that "up" in the stack means a *lower* address, these offsets will be (i) positive numbers for parameters, and (ii) negative numbers for local variables of the procedure. For example, given this code:

```
void P(int x) {
  int a, b;
  ...
}
```

and assuming that activation records do not include an access link field or space for saved registers, P's activation records will be organized as follows:

```
    _________________
   |                 |   <--- Stack Pointer
   |_________________|
b: |                 |
   |_________________|
a: |                 |   // locals have negative offsets relative to the frame pointer
   |_________________|
   | Control Link    |
   |_________________|
   | Return Addr     |   <--- Frame Pointer
   |_________________|
x: |                 |   // parameters have positive offsets relative to the frame pointe
   |_________________|
```

We will assume that each address and each integer takes up 4 bytes. Our memory-organization invariant will be that (i) the stack pointer points to the next 4-byte slot to use, and (ii) the frame pointer points to the 4-byte slot that holds the return address. (This invariant is only one possible convention that one could choose. For instance, you could choose to have the stack pointer point to the last item pushed -- i.e., the slot for b in the example above -- and you could have the frame pointer point to the control link. It is necessary to pick *some* convention and stick to it.)

For the example above, here are the offsets for each of P's parameters and locals:

- x: offset +4
- a: offset -8
- b: offset -12

The following MIPS code loads the values of a and b into registers t1 and t2, respectively:

```
lw $t1, -8($fp)   # load a
lw $t2, -12($fp)  # load b
```

To be able to generate this code at compile time (e.g., to process a statement like `x = a + b`), the offset of each local variable must be known by the compiler. Therefore, the offset of each local variable from the Frame Pointer should be stored as an attribute of the variable in the symbol table. This can be done as follows:

- Keep track of the current offset in a global variable (e.g., a static ASTnode field) or in the symboltable (i.e., add a new field to the SymTab class, with methods to set and get the field's value).
- When the symbol-table-building code starts processing a method, set the current offset to +4.
- For each parameter: add the name to the symboltable as usual, but also include the value of the current offset as an attribute, and then update the current offset by adding the size of the parameter (in bytes), which will depend on the parameter's type.
- After processing all of the parameters, set the offset to -4 (to leave room for the control link).
- For each local variable, subtract from the current offset the size of the local (in bytes), and then add the name of the local to the symboltable with the current offset as an attribute.

Note that the "current offset" is *not* reset at the start of a nested block, and thus each variable declared (somewhere) in a method M has its own, unique offset in the AR for M.

Note as well that the formal parameters are traversed in left-to-right order, and are given successively greater and greater positive offsets (with respect to the *callee's* frame pointer). Thus, we must arrange for the *caller* to evaluate the actuals from *right to left*, so that the resulting values are pushed on the stack in an order that is consistent with the numbering scheme given above.

Doubles

For MIPS code, each double variable requires 8 bytes. For example, given this code:

```
void P(double x) {
  double a, b;
  ...
}
```

variable x will be stored in the bytes at offsets +4 to +11 from the frame pointer, and the offset that should be stored in x's symboltable entry (and used to access x at runtime) is +4. Here are the offsets for all of P's locals:

- x: offset +4
- a: offset -12
- b: offset -20

Pairs of floating-point registers with consecutive numbers (e.g., {f0,f1}, {f2,f3}) must be used for double-valued computations at runtime, and there are special opcodes for operations on double values. Note that the instructions only refer to the even-numbered register of each pair (although both the even-numbered and the next-higher register both participate in the computation). For instance, the following MIPS code loads the values of a and b into the register pairs {f0,f1} and {f2,f3}, respectively:

```
l.d $f0, -12($fp)  # load a
l.d $f2, -20($fp)  # load b
```

TEST YOURSELF #1

Assume that both an address and an integer require 4 bytes of storage, and that a double requires 8 bytes. Also assume that each activation record includes parameters, a return address, a control link, and space for local variables as illustrated above. What are the offsets (in bytes) for each of the parameters and local variables in the following functions?

```
void P1(int x, int y) {
   int a, b, c;
   ...
   while (...) {
      double a, w;
   }
}

void P2() {
   int x, y;
   ...
   if (...) {
      double a;
      ...
   }
   else {
      int b, c;
```

```
        ...
    }
}
```

solution

Global Variables

As noted above, global variables are stored in the static data area. Using MIPS code, each global is stored in space labeled with the name of the variable, and the variable is accessed at runtime using its name. (Since we will be using the SPIM simulator, and since SPIM has some reserved words that can also be used as Simple variable names, you will need to add an underscore to global variable names to prevent clashes with SPIM reserved words.)

For example, if the source code includes:

```
// global variables
int g;
double d;
```

The following code would be generated to reserve space in the static data area for variables g and d:

```
        .data          # put the following in the static data area
        .align 2       # align on a word boundary
_g:     .space 4       # set aside 4 bytes
        .data
        .align 2
_d:     .space 8
```

And the following code would be generated to load the value of variable g into register t0, and to load the value of variable d into register f0:

```
lw    $t0, _g       # load contents of g into t0
l.d   $f0, _dd      # load contents of dd into f0
```

Non-Local Variables

Recall that we are using the term "non-local variable" to refer to two situations:

1. In statically-scoped languages that allow nested sub-programs, a variable can be declared in one sub-program and accessed in another, nested sub-program. Such variables are "non-local" in the nested sub-program that accesses them. These variables are stored in the activation record of the sub-program that declares them. When they are used as non-local variables, that activation record is found **at runtime** either using **access links** or a **display**

(discussed below).
2. In dynamically-scoped languages, a variable can be declared in one method, and accessed in a called method. The variable is non-local in the method that accesses it. Two different approaches to supporting the use of non-local variables in dynamically-scoped languages are discussed below.

Note that in languages (like C and C++) that permit the same name to be used in nested blocks within a method, we might also use the term "non-local variable" to refer to a use of a variable in one block that was declared in an enclosing block. However, this is not an interesting case in terms of runtime access. All of a method's variables (regardless of which block they are declared in) are stored in the method's activation record, and are accessed using offsets from the frame pointer, as discussed above.

Static Scoping

First, let's consider an example (Pascal) program that includes accesses to non-local variables (the nesting levels of the procedures are given for later reference):

```
  + program MAIN;
  | var x: integer;
  |
  |    + procedure P;
  |   (2)  write(x);
  |    +
  |
  |    + procedure Q;
  |    | var y: integer = x;
  |    |
  |    |    + procedure R;
  |    |    |   x = x + 1;
  |    |    |   y = y + x;
 (1)  (2)  (3)  if y<6 call R;
  |    |    |   call P
  |    |    +
  |    |
  |    |    call R;
  |    |    call P;
  |    |    if x < 5 call Q;
  |    +
  |
  |    x = 2;
  |    call Q;
  +
```

solution

TEST YOURSELF #2

Trace the execution of this program, drawing the activation records for the main program and for each called procedure (include only local variables and control links in the activation records). Each time a non-local variable is accessed, determine which activation record it is stored in. Notice that the relative nesting levels of the variable's declaration and its use does **not** tell you how far down the stack to look for the activation record that contains the non-local variable. In fact, this number changes for different calls (e.g., due to recursion).

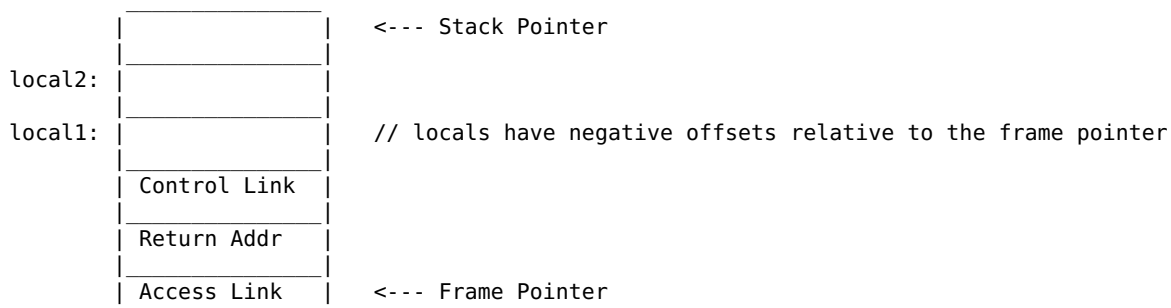
solution

Method #1: Access Links

The idea behind the use of access links is as follows:

- Add a new field to each AR -- the ***access link*** field.
- If P is (lexically) nested inside Q, then at runtime, P's AR's access link will point to the access link field in the AR of the most recent activation of Q.
- Therefore, at runtime, access links will form a chain corresponding to the nesting of sub-programs. For each use of a non-local x:
  - At compile time, use the "Level Number" attribute of x and the "Current Level Number" (of the sub-program that is accessing x) to determine how many links of the chain to follow at runtime.
  - If P at level i uses variable x, declared at level j, follow i-j links, then use x's "Offset" attribute to find x's storage space inside the AR.

We will put the access-link field as the first field in the AR (proper)—that is, register FP points to the access-link field. Thus, the layout of an AR with one parameter and two locals is as follows:



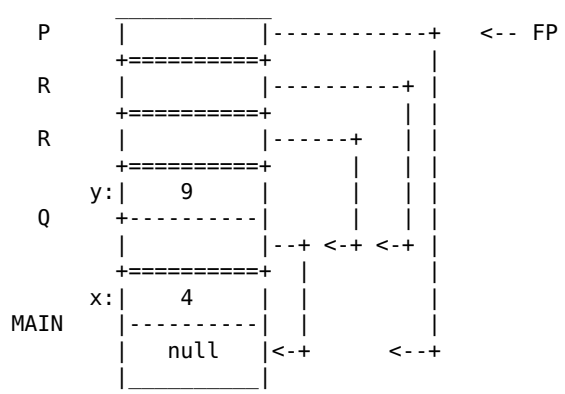
```
        |_________________|
param1: |                 |   // parameters have positive offsets relative to the frame pointe
        |_________________|
```

A good way of thinking about the ARs is in terms of linked lists: each AR is an element that can participate in two kinds of lists:

1. A list that is linked via the control-link fields (i.e., the saved FP values)
2. A list that is linked via the access-link fields

The control-link list is the ``primary list,'' because it consists of the ARs of the pending calls that have not yet completed (in the order that they need to complete). The access links provide another linked list (on typically a subset of the ARs). Note that with ordinary objects you can have exactly the same situation: each object can have multiple fields that serve to link the same set (or a subset) of objects in different linked lists.

Returning to the example program given earlier, here's a snapshot of the runtime stack after the first call from R to P (where we show only the access links and the local variables in the ARs):



To access the value of x from procedure R, access links must be followed. The number of links that must be followed is:

(level # of R) - (level # of decl of x)
= 3 - 1 = 2

So the code for `y = x` in procedure R would be as shown below (Note: we assume that because we have access links, FP always points to the access-link field.) We also assume that both variable x and variable y are at offset -12 in their respective ARs. Because the access link is at offset 0 with respect to FP, the return address is at offset -4, and the control link is at -8. Because x and y are the first local variables in their respective program/procedure, they are each at

offset -12 in their respective ARs.)

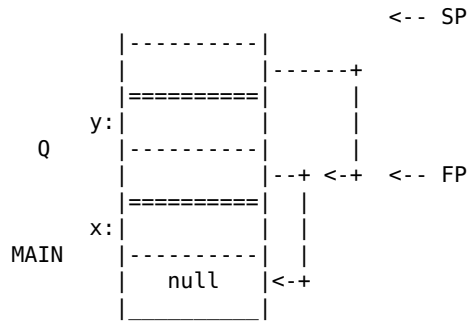
```
         move $t0, FP     // no links followed yet; t0 holds ptr to R's access link field
         lw $t0, ($t0)     // 1 link: t0 holds ptr to Q's AR's access link field
 +->     lw $t0, ($t0)     // 2 links: t0 holds ptr to main's AR's access link field
 |       lw $t0, -12($t0)  // t0 holds value of x
 |       sw $t0, -12(FP)  // y = x
 |
This code would be repeated if we needed to follow more links.
```

How to set up access links:

- The link is set up by the *calling* procedure.
- How to set up the link depends on the relative nesting levels of the calling and called procedures.

There are two cases:
Case 1: The calling procedure's level is less than the called procedure's level (i.e., the called procedure is nested ***directly*** inside the calling procedure, because if it's not, it's invisible and it can't be called). In this case, the called procedure's access link should point to the access link field of the calling procedure's AR. This case occurs in the example when Q calls R.

In this case, the calling procedure sets up the called procedure's access link by pushing the value of the FP just before the call, since the FP points to its access link. For example, when Q is calling R, here is a picture of the stack just after R's access link has been set up by Q:



Case 2: The calling procedure's level is greater than or equal to the called procedure's level. In this case, the called procedure's access link should point to an AR that is already on the stack. The value for the new access link is found by starting with the value of the calling procedure's access link field (which is pointed to by the FP), and following X-Y access links, where:

X = calling procedure's level
Y = called procedure's level

The following code loads the value of the calling

procedure's access link field into register t0:

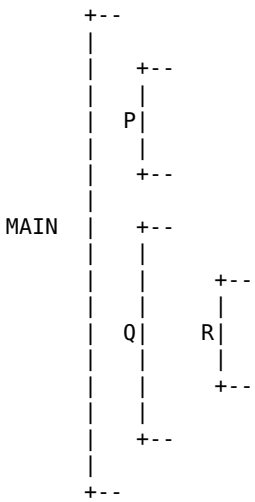
```
  lw    $t0, 0(FP)
```

If X == Y (i.e., no links should be followed to find the value of the new access link), then the value of `t0` should simply be pushed onto the stack. If X is greater than Y, then the code:

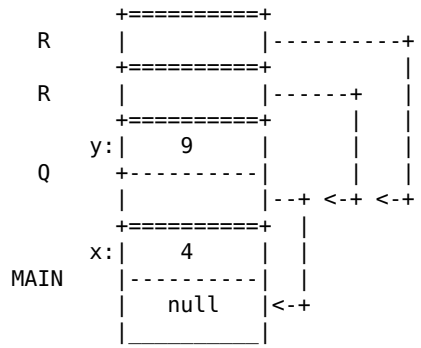
```
  lw    $t0, 0($t0)
```

should be generated X - Y times, before pushing the value of `t0`.

To illustrate this situation, consider two cases from the example code: R calls P, and Q calls itself. Recall that the nesting structure of the example program is:



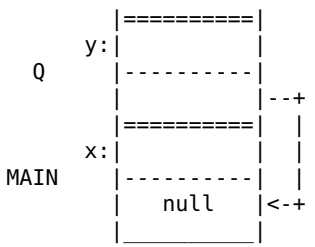
When R is about to call P, the stack looks like this:



Since P is nested inside MAIN, P's access link should point to MAIN's AR (i.e., the bottom AR on the stack). R's nesting level is 3 and P's nesting level is 2. Therefore, we start with the value of R's access link (the pointer to Q's AR) and follow *one* link. This retrieves the values of Q's access link, which is (as desired) a pointer to MAIN's AR. This is the value that will be pushed onto the stack (copied into P's AR as its access

link).

When Q is about to call itself, the stack looks like this:



The access link in the new AR for Q should be the same as the access link in the current AR for Q; namely, it should be a pointer to the AR for MAIN. This value is found by starting with the value of Q's access link (a pointer to MAIN's AR) and following *zero* links (because X = Y = 1).

TEST YOURSELF #3

Trace the execution of the example program again. Each time a procedure is called, determine which case applies in terms of how to set up the called procedure's access link. Then use the appropriate algorithm to find the value of the new access link, and draw the new AR with its access link.

Each time a non-local variable x is accessed, make sure that you find its activation record by following i - j access links (where i is the nesting level of the procedure that uses x and j is the nesting level of the procedure that declares x).

solution

Access Links: Summary

To ***use*** an access link:

> Follow i - j links to find the AR with space for non-local x, where i is the nesting level of the procedure that uses x and j is the nesting level of the procedure that declares x.

To ***set up*** an access link:

- case #1: Calling procedure's level is less than called procedure's level: Push the value of the FP to be the access link field of the called procedure's AR.
- case #2: Calling procedure's level is greater

than or equal to the called procedure's level: Find the value to push by starting with the value of the calling procedure's access links, then following X-Y links, where X = calling procedure's level, and Y = called procedure's level.

Method #2: The Display

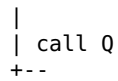
The motivation for using a display is to avoid the runtime overhead of following multiple access links to find the activation record that contains a non-local variable. The idea is to maintain a global "array" called the ***display***. Ideally, the display is actually implemented using registers (one for each array element) rather than an actual array; however, it can also be implemented using an array in the static data area.

The size of the display is the maximum nesting depth of a procedure in the program (which is known at compile time). The display is used as follows:

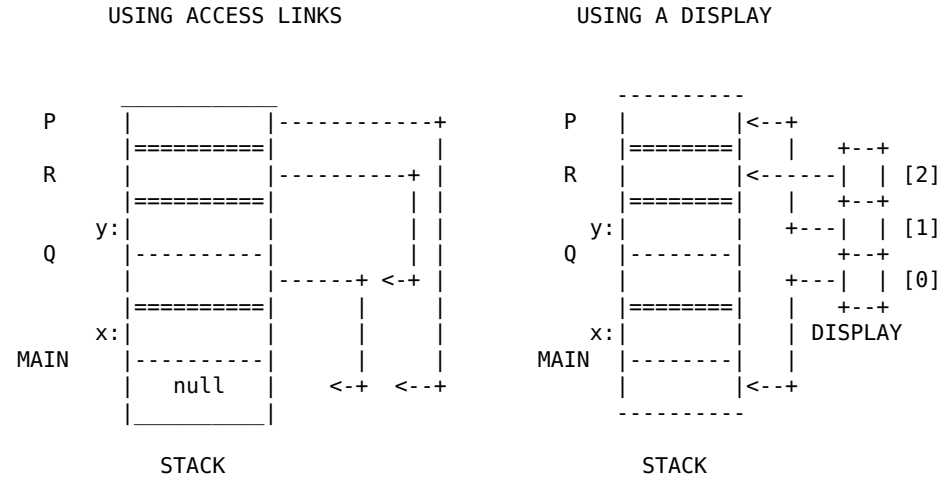
- When procedure P at nesting level k is executing, DISPLAY[0],...DISPLAY[k-2] hold pointers to the ARs of the most recent activations of the k-1 procedures that lexically enclose P. DISPLAY[k-1] holds a pointer to P's AR.
- To access a non-local variable declared at level x, use DISPLAY[x-1] to get to the AR that holds x, then use the usual offset to get x itself.

To illustrate this, refer back to our running example program, outlined below:

```
      +--
      |int x;
      |
      |   +--
      |   |
      |  P|...x...
      |   |
      |   +--
      |
MAIN  |   +--
      |   | int y;
      |   |
      |   |     +--
      |   |     |...x...y...
      |  Q|    R|call R
      |   |     |call P
      |   |     +--
      |   |
      |   | call R
      |   | call P
      |   | if (...) call Q
      |   +--
```



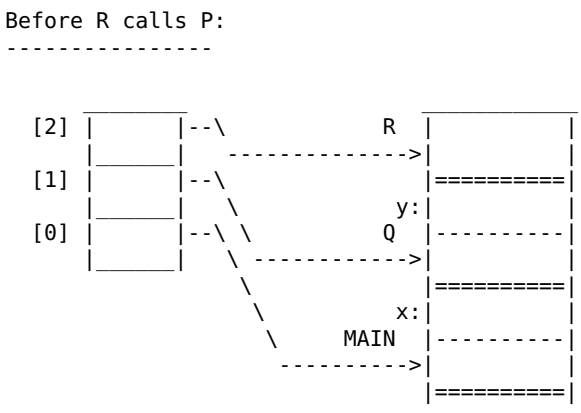
Below are two pictures comparing the use of access links with the use of a display. Both show the same moment at runtime.

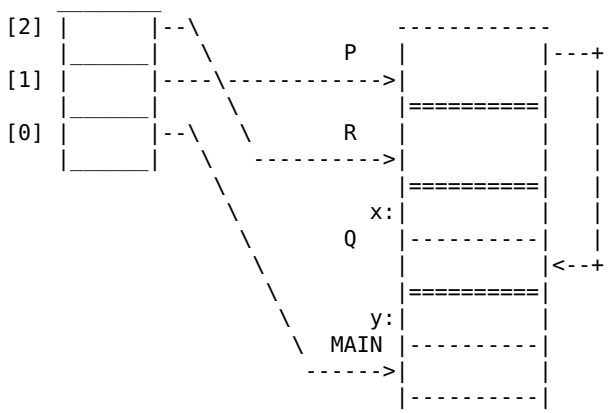


To maintain the display, a new field (called the "save-display" field) is added to each activation record. The display is maintained as follows:

- When a procedure at nesting level k is called:
  - The current value of DISPLAY[k-1] is saved in the "save-display" field of the new AR.
  - DISPLAY[k-1] is set to point to (the save-display field of) the new AR.
- When the procedure returns, DISPLAY[k-1] is restored using the value saved in the "save-display" field (of the returning procedure).

This process is illustrate below, showing how the display and the save-display fields are updated when R calls P (only the local variable and save-display fields of the ARs are shown).





TEST YOURSELF #4

Trace the execution of the running example program, assuming that a display is used instead of access links. Each time a non-local variable x is accessed, make sure that you understand how to find its AR using the display.

Comparison: Access Links vs The Display

- Access links can require more time (at runtime) to access non-locals (especially when the non-local is many nesting levels away).
- It can also require more time to set up a new access link when a procedure is called (if the nesting level of the called procedure is much smaller than the nesting level of the calling procedure).
- Displays require more space (at runtime).
- If the compiler is flexible enough to implement the display using registers if enough are available, or using space in the static data area, then the compiler code itself may be rather complicated (and error-prone).
- In practice, sub-programs usually are not very deeply nested, so the runtime considerations may not be very important.

Dynamic Scoping

Recall that under dynamic scoping, a use of a non-local variable corresponds to the declaration in the "most recently called, still active" method. So the question of which non-local variable to use can't be determined at compile time. It can only be determined at run-time. There are two ways to implement access to non-locals under dynamic scope: "deep access"

and "shallow access", described below.

Deep Access

Using this approach, given a use of a non-local variable, control links are used to search back in the stack for the most recent AR that contains space for that variable. Note that this requires that it be possible to tell which variables are stored in each AR; this is more natural for languages that are *interpreted* rather than being compiled (which was indeed the case for languages that used dynamic scope). Note also that the number of control links that must be followed cannot be determined at compile time; in fact, a different number of links may be followed at different times during execution, as illustrated by the following example program:

```
void P() { write x; }

void Q() {
  x = x + 1;
  if (x < 23) Q();
  else P();
}

void R() {
  int x = 20;
  Q();
  P():
}

void main() {
  int x = 10;
  R();
  P();
}
```

TEST YOURSELF #5

Trace the execution of the program given above. Note that method P includes a use of non-local variable x. How many control links must be followed to find the AR with space for x each time P is called?

Shallow Access

Using this approach, space is allocated (in registers or in the static data area) for every variable name that is in the program (i.e., one space for variable x even if there are several declarations of x in different methods). For every reference to x, the generated code refers to the *same* location.

When a method is called, it saves, in its own AR, the current values of all of the variables that it declares itself (i.e., if it declares x and y, then it saves the values of x and y that are currently in the space for x and y). It restores those values when it finishes.

Note that this means that when a method accesses a *non-local* variable x, the value of x from the most-recently-called, still-active method is stored in the (single) location for x. There is no need to go searching down the stack!

TEST YOURSELF #6

Question 1: Trace the execution of the program given above again, this time assuming that shallow access is used.

Question 2: What are the advantages and disadvantages of shallow access compared with deep access? (Consider both time and space requirements.)

Contents

- Overview
- Spim
- Auxiliary Fields and Methods
- Code Generation for Global Variable Declarations
- Code Generation for Functions

- Function Preamble
- Function Entry
- Function Body
- Function Exit

- Code Generation for Statements

- Write Statement
- If-Then Statement
  - Test Yourself #1
- Return Statement
- Read Statement
- Digression: Code Generation for IdNodes
  - genJumpAndLink
  - codeGen
  - genAddr
- Assignment Statement

- Code Generation for Expressions

- Assign
- Literals
- Function Call
- Non Short-Circuited Operators
- Short-Circuited Operators
  - Test Yourself #2

- Control-Flow Code

- Test Yourself #3
- Test Yourself #4

Overview

Code can be generated by a syntax-directed translation while parsing or by traversing the abstract syntax tree after the parse (i.e., by writing a codeGen method for the appropriate kinds of AST nodes). We will assume the latter approach, and will discuss code generation for a subset of the C language. In particular, we will discuss generating MIPS assembly code

suitable for input to the Spim interpreter. Some information on Spim is provided in the next section; the following sections discuss code generation for:

- global variables
- functions (entry and exit)
- statements
- expressions

Spim

Documentation on Spim is available on-line:

- Reference Manual (pdf)
- Instructions for download and install (html)

To run the (plain) Spim interpreter, type:

```
spim -file <name>
```

where `<name>` is the name of a file that contains MIPS assembly code (the file produced by your compiler). This will cause Spim to process the code in the file; if there are syntax errors, they will be reported, and the code will not execute. Otherwise, the code will execute; output will be printed to your terminal, and you will get error messages for any run-time errors that result.

To run Spim with an X-windows interface (on a Linux machine), type just: `xspim`. This will cause a window to open. Click on the `load` button in that window, then type in the name of your assembly-code file, then press `return`. If there are syntax errors, you will see error messages. If there are no errors, you can run your program by clicking on `run`, then (in the small window that will be opened) on `OK`. If your program generates any output, a new window will be opened to display that output.

Spim uses the following special registers (there are others; you can see the on-line Spim documentation for those, but you should not need them for the class project):

Register		**Purpose**
$sp		stack pointer
$fp		frame pointer

$ra		return address
$v0		used for system calls and to return int values from function calls, including the syscall that reads an int
$f0		used to return double values from function calls, including the syscall that reads a double
$a0		used for output of int and string values
$f12		used for output of double values
$t0 - $t7		temporaries for ints
$f0 - $f30		registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)

Auxiliary Constants and Methods

To simplify the task of code generation, it is convenient to have a set of constants (final static fields) that define the string representations of the registers that will be used in the generated code and the values used to represent *true* and *false*, as well as a set of methods for actually writing the generated code to a file. We will assume that we have the following register constants: SP, FP, RA, V0, A0, T0, T1, F0, F12, as well as the constants TRUE and FALSE (and that TRUE is represented as 1, and false as 0). We will also assume that we have the following methods:

Method	**Purpose**
generate	write the given op code and arguments, nicely formatted, to the output file
generateIndexed	the arguments are: an op code, a register R1, another register R2, and an offset; generate code of

	the form: `op R1, offset(R2)`
genPush(String reg, int bytes)	generate code to push the value in the given register onto the stack; parameter `bytes` is 4 for an int and 8 for a double
genPop(String reg, int bytes)	generate code to pop the top-of-stack value into the given register
nextLabel	return a string to be used as a label (more on this later)
genLabel	given a label L, generate: `L:`

Code Generation for Global Variable Declarations

For each global variable `v`, generate:

```
        .data
        .align 2  # align on a word boundary
    _v: .space N
```

where `N` is the size of the variable in bytes. (Scalar integer variables require 4 bytes; double variables require 8 bytes.) This code tells the assembler to set aside N bytes in the static data area, in a location labeled with the name `_v`.

Example: Given this source code:

```
        int x;
        double y;
```

you should generate this code:

```
        .data
        .align 2
    _x: .space 4
        .data
        .align 2
    _y: .space 8
```

It is not actually necessary to generate `.data` if the previous generated code was also for a global variable declaration; however, since function declarations can be intermixed with

global variable declarations (and cause code to be generated in the text area, not the static data area), this may not be the case; it is safe (and easier) just to generate those directives for every global variable.

Code Generation for Functions

For every function you will generate code for:

- the function "preamble"
- the function entry (to set up the function's Activation Record)
- the function body (its statements)
- function exit (restoring the stack, and returning to the caller).

Function Preamble

For the `main` function, generate:

```
        .text
        .globl main
  main:
```

For all other functions, generate:

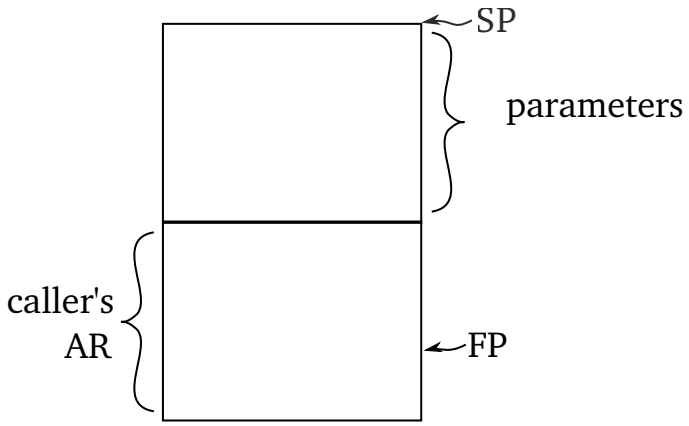
```
        .text
        _<functionName>:
```

using the actual name in place of `<functionName>`. This tells the assembler to store the following instructions in the text area, labeled with the given name.

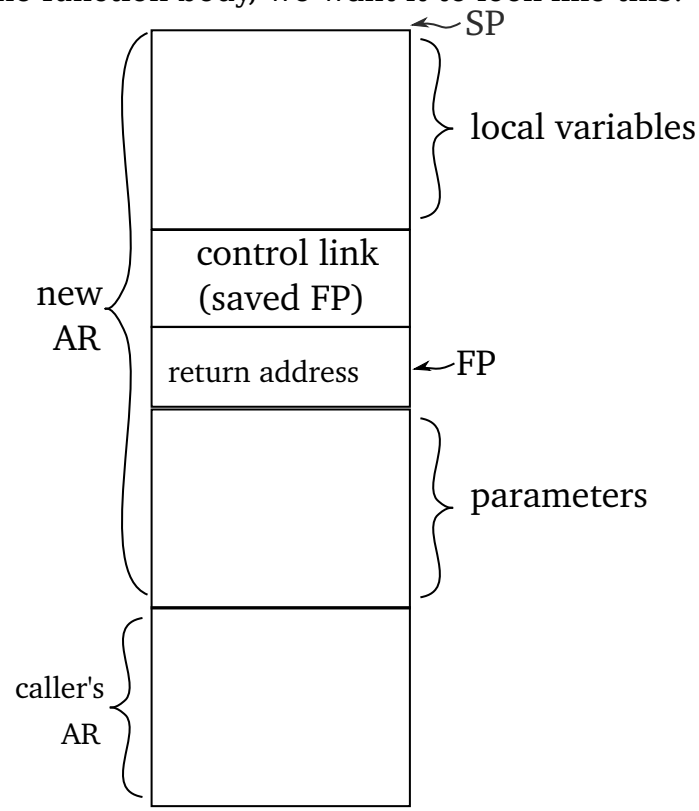
After generating this "preamble" code, you will generate code for (1) function entry, (2) function body, and (3) function exit.

Function Entry

We assume that when a function starts executing, the stack looks like this:



Before starting to execute the statements in the function body, we want it to look like this:



The parameters will already be on the stack (pushed by calling function). So the code for function entry must do the following:

1. push the return address
2. push the control link
3. set the FP
4. push space for local variables

Here's the code you need to generate:

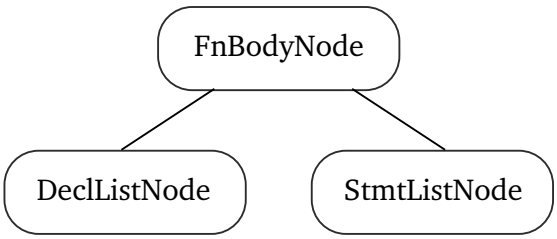
```
# (1) Push the return addr
  sw   $ra, 0($sp)
  subu $sp, $sp, 4
# (2) Push the control link
  sw   $fp, 0($sp)
  subu $sp, $sp, 4
# (3) set the FP
# Note: our convention for $sp is that it points to the first unused word of the stack.
# The reason for adding 8 is that the unused word and the control link each take 4 bytes
```

```
    addu $fp, $sp, 8
  # (4) Push space for the locals
    subu $sp, $sp, <size of locals in bytes>
```

Note: `<size of params>` and `<size of locals>` will need to be available to the code generator. The symbol-table entry for the function name will have information about the parameters (because that will have been used for type checking). For example, it might have a list of the symbol-table entries for the parameters. You could also store the total size of the parameters in the function name's symbol-table entry, or you could write a method that takes the list of parameters as its argument and computes the total size. It is not so easy to compute the total size of the local variables at code-generation time; it is probably a better idea to do that during name analysis. The name-analysis phase will be computing the offsets for the parameters and local variables anyway; it should not be difficult to extend that code to also compute the total size of the locals (and to store that information in the function name's symbol-table entry).

Function Body

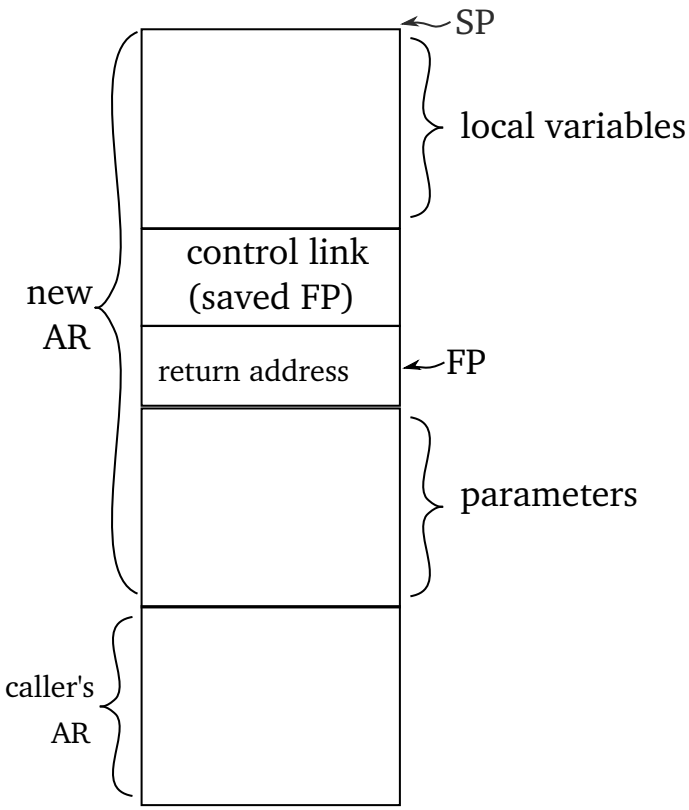
Note: we are talking about the codeGen method for the FnBodyNode, whose subtree will look like this:



There is no need to generate any code for the declarations. So to generate code for the function body, just call the codeGen method of the StmtListNode, which will in turn call the codeGen method of each statement in the list. What those methods will do is discussed below in the section on Code Generation for Statements.

Function Exit

Just before a function returns, the stack looks like this:



We need to generate code to pop off this function's AR, then to jump to the address in the "return address" field. Popping off the AR means restoring the SP and FP. Note that we want to move the SP to where the FP is currently pointing, but if there may be an interrupt that could use the stack, we don't want to change the SP until we're finished with all of the values in the current AR (in particular, the control link, which is used to restore the FP). Therefore, we use a temporary register (t0) to save the address that is initially in the FP

Here is the code that needs to be generated:

```
lw   $ra, 0($fp)      # load return address
move $t0, $fp         # FP holds the address to which we need to restore SP
lw   $fp, -4($fp)     # restore FP
move $sp, $t0         # restore SP
jr   $ra              # return
```

Note that there are two things that cause a function to return:

1. A `return` statement is executed, or
2. The last statement in the function is executed (i.e., execution "falls off the end" of the function).

You could generate the "return" code given above for each `return` statement as well as after the last statement in the function body. A more space-efficient approach would be:

- Generate the "return" code just once after generating the code for the function body. Label that code with a unique label (e.g., the result of calling nextLabel).
- For each `return` statement, generate a jump to the label you used (the op code for an unconditional jump is just `b`).

What about a return statement that returns a value? As discussed below, the codeGen method for the returned expression will generate code to evaluate that expression, leaving the value on the stack. The MIPS convention is to use register V0 to return a int value from a function and to use register F0 to return a double value. So the codeGen method for the return statement should generate code to pop the value from the stack into the appropriate register (before generating the "return" code or the jump to the return code discussed above).

Code Generation for Statements

You will write a different codeGen method for each kind of StmtNode. You are strongly advised to write this method for the WriteIntStmtNode, WriteDblStmtNode, and WriteStrStmtNode first. Then you can test code generation for the other kinds of statements and the expressions by writing a program that computes and prints a value. It will be much easier to find errors in your code this way (by looking at the output produced when a program is run) than by looking at the assembly code you generate.

Write Statement

To generate code for a write statement whose expression is of type *int* you must:

1. Call the codeGen method of the expression being printed. That method will generate code to evaluate the

expression, leaving that value on the top of the stack.
2. Generate code to pop the top-of-stack value into register A0 (a special register used for output of strings and ints)
3. Generate code to set register V0 to 1.
4. Generate a `syscall` instruction.

Below is the code *you* would write for the codeGen method of the WriteIntStmtNode.

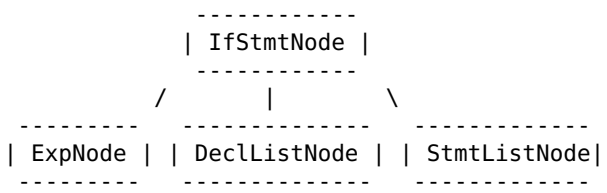
```
// step (1)
myExp.codeGen();

// step (2)
genPop(A0, 4);

// step (3)
generate("li", V0, 1);

// step (4)
generate("syscall");
```

The code for the WriteStrStmtNode and the WriteDblStmtNode is similar, except for the following:

- For a string, the codeGen method of the expression being printed will leave the address of the string on the stack.
- For a double, the value to be written must be popped into register F12 instead of A0.
- For a string, register V0 must be set to 4.
- For a double, register V0 must be set to 3.

If-Then Statement

The AST for an if-then statement looks like:



There are two different approaches to generating code for statements that involve conditions (e.g., for if statements and while loops):

1. The *numeric* method, and
2. the *control-flow* method.

We will discuss code generation for if-then statements assuming the numeric method here; the control-flow method will be discussed later. The code generated by the

IfStmtNode's codeGen method will have the following form:

1. Evaluate the condition, leaving the value on the stack.
2. Pop the top-of-stack value into register T0.
3. Jump to FalseLabel if T0 == FALSE.
4. Code for the statement list.
5. FalseLabel:

Labels

Note that the code generated for an if-then statement will need to include a label. Each label in the generated code must have a unique name (although we will refer to labels in these notes using names like "FalseLabel" as above). As discussed above, we will assume that there is a method called *nextLabel* that returns (as a String) a new label every time it is called, and we will assume that there is a method called *genLabel* that prints the given label to the assembly-code file.

TEST YOURSELF #1

Question 1: What is the form of the code generated by an IfElseStmtNode's codeGen method?

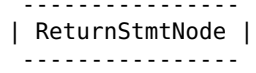
Question 2: What is the actual code that needs to be written for the IfStmtNode's codeGen method?

Question 3: What is the form of the code generated by a WhileStmtNode's codeGen method?

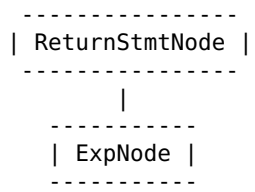
solution

Return Statement

The AST for a return statement is either:



or:



As discussed above, if a value is being returned, the ReturnStmtNode's codeGen method should call its ExpNode's codeGen method (to generate code to evaluate the returned expression, leaving the value on the stack), then should generate code to pop that value into register V0 or register F0 (depending on its type).

To generate the code that actually does the return, use one of the following approaches:

1. For each return statement in the program, generate a copy of the code that pops the AR off the stack then jumps to the return address (that code was discussed above under Function Exit), or
2. For each return statement in the program, generate a jump to the "return" code that is generated at the end of the function. Note that in this case you will need to label that return code, and you will need to know what that label is when generating code for a return statement.

Read Statement

The AST for a read statement is one of the following:

To read an integer value into register V0, you must generate this code:

```
     li    $v0,  5
  syscall
```

The code loads the special value 5 into register V0, then does a syscall. The fact that V0 contains the value 5 tells the syscall to read an integer value from standard input,

storing the value back into register V0. So we must start by generating the above code, then we must generate code to store the value from V0 to the address of the IdNode.

To read a double value, load the value 7 into V0, do a syscall, and expect the input value to be in register F0 (not V0).

Digression: Code Generation for IdNodes

Before considering other kinds of statements, let's think about the role identifiers will play in code generation. Names show up in the following contexts:

- function calls (the name of the called function)
- expressions (an expression can be just a name, or a name can be the operand of any operator)
- assignment statements (on the left-hand side)

The code that needs to be generated for the name will be different in each context:

1. For a function call, we will need to generate a jump-and-link instruction using the name of the function (the same name that was generated as a label in the function's "preamble" code).
2. For an expression, we will need to generate code to fetch the current value either from the static data area or from the current Activation Record, and to push that value onto the stack.
3. For an assignment, we will need to generate code to push the *address* of the variable (either the address in the static data area, or in the current Activation Record) onto the stack. Then we will generate code to store the value of the right-hand-side expression into that address.

Therefore, it seems reasonable to write three different code-generation methods for the IdNode class; for example:

- genJumpAndLink,

- codeGen,
- genAddr.

We use "codeGen" for the second case (fetching the value and pushing it onto the stack) since that is what the codeGen methods of all ExpNodes must do.

genJumpAndLink

The genJumpAndLink method will simply generate a jump-and-link instruction (with opcode `jal`) using the appropriate label as the target of the jump. If the called function is "main", the label is just "main". For all other functions, the label is of the form:

_<functionName>

codeGen

The codeGen method must copy the value of the global / local variable into a register (e.g., T0 for an int variable and F0 for a double), then push the value onto the stack. Different code will be generated for a global and a local variable. Below are four examples.

```
lw $t0 _g       // load the value of int
                global g into T0

lw t00(fp)      // load the value of the int
                local stored at offset 0
                into T0

l.d $f0 _d      // load the value of dbl
                global d into F0

l.d
f0 – 4(fp)      // load the value of the dbl
                local stored at offset -4
                into F0
```

Note that this means there must be a way to tell whether an IdNode represents a global or a local variable. There are several possible ways to accomplish this:

- The symbol-table entry includes a "kind" field (which distinguishes between globals and locals).
- Different sub-classes of the Sym class are used for globals and for local variables (so you can tell whether you have a global or a local using "instanceof", or using an IsGlobal method that you write for each sub-

class of Sym).
- The symbol-table entry includes an "offset" field; for local variables, that field has a value less than or equal to zero, while for globals, the value is greater than zero.

genAddr

The genAddr method must load the address of the identifier into a register (e.g., T0), then push it onto the stack. The code is very similar to the code to load the *value* of the identifier; we just use the *la* (load address) opcode instead of the *lw* (load word) or *l.d* (load double) opcode.

Here is the code you need to generate to load the address of a global variable *g* into register T0 (this works whether `g` is int or double, since an address always takes 4 bytes):

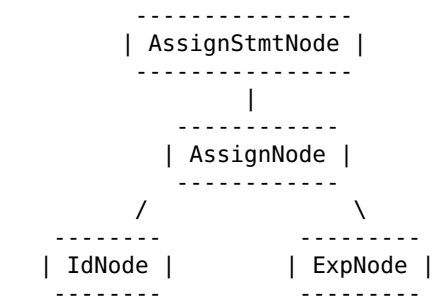
```
la $t0, _g
```

and here is the code for a local, stored at offset -8:

```
la t0,−8(fp)
```

Assignment Statement

The AST for an assignment statement looks like this:



The AssignStmtNode's codeGen method can call the AssignNode's method to do the assignment, but be careful: an AssignNode is a subclass of ExpNode, so like all nodes that represent expressions, it must leave the value of the expression on the stack. Therefore, the AssignStmtNode must generate code to pop (and ignore) that value.

Code Generation for

Expressions

The codeGen method for the subclasses of ExpNode must all generate code that evaluates the expression and leaves the value on top of the stack. We have already talked about how to do this for IdNodes; in the subsections below we discuss code generation for other kinds of expressions.

Assign

The codeGen method for an assignment expression must generate code to:

1. Evaluate the right-hand-side expression, leaving the value on the stack.
2. Push the address of the left-hand-side Id onto the stack.
3. Store the value into the address.
4. Leave a copy of the value on the stack.

Most of the work is done by calling the AssignNode's children's methods: the codeGen method of the right-hand-side ExpNode, and the genAddr method of the left-hand-side IdNode. How to accomplish the rest of the assignment is left to you to figure out (it isn't too difficult).

Literals

The codeGen methods for IntLitNodes must simply generate code to push the literal value onto the stack. The generated code will look like this:

```
li    $t0, <value>        # load value into T0
sw    $t0, ($sp)          # push onto stack
subu  $sp, $sp, 4
```

The code for a DblLitNode is similar, except that the value must be loaded into a double register using the appropriate opcode, and the "push" code is different for a double value. For example:

```
li.d  $f0, <value>    # load value into F0
s.d   $f0, -4($sp)    # push onto stack
subu  $sp, $sp, 8
```

For a StringLitNode, the string literal itself

must be stored in the static data area, and its *address* must be pushed. The code to store a string literal in the static data area looks like this:

```
          .data
<label>: .asciiz <string value>
```

Note:

1. <label> needs to be a *new* label; e.g., returned by a call to nextLabel.
2. The <string value> needs to be a string in quotes. You should be storing string literals that way, so just write out the value of the string literal, quotes and all.

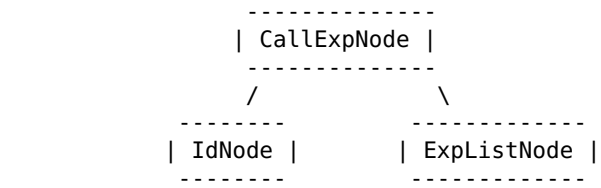
To avoid storing the same string literal value more than once, keep a hashtable in which the keys are the string literals, and the associated information is the static-data-area label. When you process a string literal, look it up in the hashtable: if it is there, use its associated label; otherwise, generate code to store it in the static data area, and add it to the hashtable.

The code you need to generate to push the address of a string literal onto the stack looks like this:

```
.text
la   $t0, <label>       # load addr into $t0
sw   $t0, ($sp)         # push onto stack
subu $sp, $sp, 4
```

Function Call

The AST for a function call looks like:



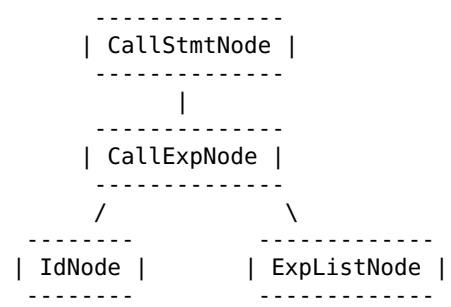
We need to generate code to:

1. Evaluate each actual parameter, pushing the values onto the stack;
2. Jump and link (jump to the called function, leaving the return address in the RA register).
3. Push the returned value (which will be in

register V0 or F0 depending on the return type) onto the stack.

Since the codeGen method for an expression generates code to evaluate the expression, leaving the value on the stack, all we need to do for step 1 is call the codeGen method of the ExpListNode (which will in turn call the codeGen methods of each ExpNode in the list). For step 2, we just call the genJumpAndLink method of the IdNode. For step 3, we just call genPush(V0, 4) or genPush(F0, 8).

Note that there is also a call *statement*:



In this case, the called function may not actually return a value (i.e., may have return type *void*). It doesn't hurt to have the CallExpNode's codeGen method push the value in V0 (or F0) after the call (it will just be pushing some random garbage), but it *is* important for the CallStmtNode's codeGen method to pop that value.

Non Short-Circuited Operators

In general, the codeGen methods for the non short-circuited operators (PlusNode, MinusNode, ..., NotNode, LessNode, ..., EqualsNode, etc.) must all do the same basic sequence of tasks:

1. Call each child's codeGen method to generate code that will evaluate the operand(s), leaving the value(s) on the stack.
2. Generate code to pop the operand value(s) off the stack into register(s) (e.g., T0 and T1 for ints; F0 and F2 for doubles). Remember that if there are two operands, the *right* one will be on the top of the stack.

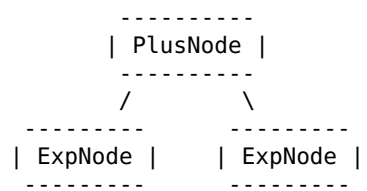
3. Generate code to perform the operation (see Spim documentation for a list of opcodes).
4. Generate code to push the result onto the stack.

The relational and equality operators are more complicated if they are applied to double operands (because there are no opcodes that "directly" compute the desired result). However, implementing those cases is not required for the class project (unless you want extra credit), so we will not discuss the details here.

To minimize the amount of code you must write, you might want to define subclasses of ExpNode that have very similar code-generation methods. For example, you could define a BinaryExpNode class whose codeGen method does all of the steps above, calling an opCode method (that would be implemented in each BinaryExpNode subclass) to get the appropriate opcode for use in step 3.

Note that the comparison operators (described in Section 2.6 of the Spim Reference Manual) produce one when the result of the comparison is true. However, the not operator does a bitwise logical negation, so it will not work correctly if you use 0 and 1 to represent true and false. Thus, it is better to use the seq operator instead of the not operator.

Example: Recall that the AST for an addition looks like this:

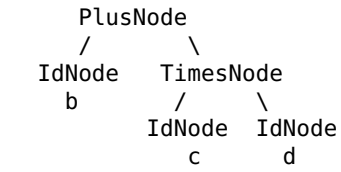


Here is the codeGen method for the PlusNode, assuming that both operands are ints.

```
public void codeGen() {
    // step 1: evaluate both operands
    myExp1.codeGen();
    myExp2.codeGen();

    // step 2: pop values in T0 and T1
    genPop(T1, 4);
    genPop(T0, 4);
```

```
    // step 3: do the addition (T0 = T0 + T1)
    generate("add", T0, T0, T1);

    // step 4: push result
    genPush(T0, 4)
}
```

To illustrate how code is generated for an expression involving several operators, consider generating code for the expression: `b + c * d`. Here is the AST for the expression:



Below is the sequence of calls that would be made at compile time to generate code for this expression, and a description of what the generated code does.

```
       Sequence of calls          What the generated code does
       -----------------          ----------------------------

+--- PlusNode.codeGen()
|      IdNode.codeGen()  --------->       push b's value
| +-   TimesNode.codeGen()
| |      IdNode.codeGen() --------->      push c's value
| |      IdNode.codeGen() --------->      push d's value
| |
| +------------------------------->   pop d's value into T1
|                                      pop c's value into T0
|                                      T0 = T0 * T1
|                                      push T0's value
|
+--------------------------------> pop result of * into T1
                                   pop b's value into T0
                                   T0 = T0 + T1
                                   push T0's value
```

Short-Circuited Operators

The short-circuited operators are represented by AndNodes and OrNodes. "Short-circuiting" means that the right operand is evaluated only if necessary. For example, for the expression `(j != 0) && (k/j > epsilon)`, the sub-expression `(k/j > epsilon)` is evaluated only if variable `j` is not zero. Therefore, the code generated for an AndNode must work as follows:

```
evaluate the left operand
if the value is true then
   evaluate the right operand;
   that value is the value of the whole expression
```

```
else
   don't bother to evaluate the right operand
   the value of the whole expression is false
```

Similarly, for an OrNode:

```
evaluate the left operand
if the value is false then
   evaluate the right operand;
   that value is the value of the whole expression
else
  don't bother to evaluate the right operand
  the value of the whole expression is true
```

This means that the code generated for the logical operators will need to involve some jumps depending on the values of some expressions.

TEST YOURSELF #2

Expand the outlines given above for the code generated for AndNodes and OrNodes, giving a lower-level picture of the generated code. Use the outline of the code generated for an if-then statement as a model of what to write.

solution

Control-Flow Code

As mentioned above in the section on If-Then Statements, there are actually two different approaches to generating code for statements with conditions (like if statements and while loops):

1. The *numeric* approach: This is the approach that we have assumed so far. Using the numeric approach, the codeGen method for a statement with a condition generates code to evaluate the condition, leaving the value on the stack. That value is then popped, and a jump is executed if it has a particular value.
2. The *control-flow* or *jump-code* approach: In this case, the code-generation method for the condition has two parameters, both of which are labels (i.e., Strings) named TrueLabel and FalseLabel. Instead of leaving the value of the condition on the stack, the code generated to evaluate the condition jumps to the TrueLabel if the condition is

true, and jumps to the FalseLabel if the condition is *false*.

We will assume that the new code-generation method for the condition is called genJumpCode. As we will see, the reason to prefer the control-flow approach over the numeric approach is that, using the control-flow approach, we will generate fewer instructions for statements that involve conditions (so the generated code will be smaller and will run faster).

First, let's reconsider code generation for an if-then statement; this time we'll use the control-flow method instead of the numeric method for evaluating the condition part of the statement. Under this new assumption, the code generated by the IfStmtNode's codeGen method will have the following form:

```
          -- code to evaluate the condition, jumping to TrueLab if it is true,
             and to DoneLab if it is false --
TrueLab:
          -- code for the statement list --
DoneLab:
```

The actual code written for the IfStmtNode's codeGen method will be:

```
public void codeGen() {
   String trueLab = nextLabel();
   String doneLab = nextLabel();
   myExp.genJumpCode(trueLab, doneLab);
   genLabel(trueLab);
   myStmtList.codeGen();
   genLabel(doneLab);
}
```

To implement the control-flow approach, in addition to changing the codeGen method for the statements that have conditions, we must also write a new genJumpCode method for each AST node that could represent a condition. Below, we look at two representative cases, for IdNode and LessNode. We give the code generated for the (old) numeric approach, and for the (new) control-flow approach so that we can see which code is better (in terms of the number of instructions). (The code given below assumes that all operands are int, and it is not quite assembly code -- for example, for clarity, we use "push" instead of the actual two instructions that implement a push operation.)

```
IdNode:  numeric                          control-flow
```

```
-------                              ------------

 lw  $t0, <var's addr>               lw  $t0, <var's addr>
 push $t0                            beq $t0, FALSE, falseLab
                                     b   trueLab
```

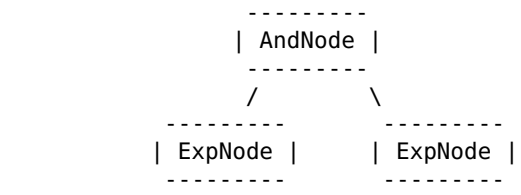
Note that in both approaches to generating code for an IdNode, 3 instructions are generated (because "push" is actually 2 instructions). However, while all 3 of the numeric-approach instructions will always execute, the last instruction generated by the control-flow approach will execute only if the value of the Id is *true*.

```
LessNode:  numeric                                  control-flow
           -------                                  ------------

           -- code to evaluate both operands        -- ditto
           -- pop values into T1, T0                -- ditto
           slt  $t2, $t0, $t1                       blt $t0, $t1, trueLab
           push $t2                                 b   falseLab
```

The operands of a LessNode are integer expressions, not boolean expressions, so both approaches start by generating (the same) code to evaluate the operands and to pop their values into registers T0 and T1. After that, however, the numeric approach will generate 3 instructions (to set $t2 to TRUE or FALSE as appropriate, then to push that value onto the stack -- remember that "push" is really two instructions), while the control-flow code will generate only 2 instructions. Furthermore, as was the case for the IdNode, all three of the numeric-approach instructions will always execute, while the last instruction generated by the control-flow approach will only execute if the comparison evaluates to FALSE.

Now let's consider how to write the new genJumpCode method for the short-circuited operators (AndNodes and OrNodes).

Recall that the AST for an && expression looks like this:



Here's how the genJumpCode method of the AndNode works:

- Start by calling the genJumpCode method of

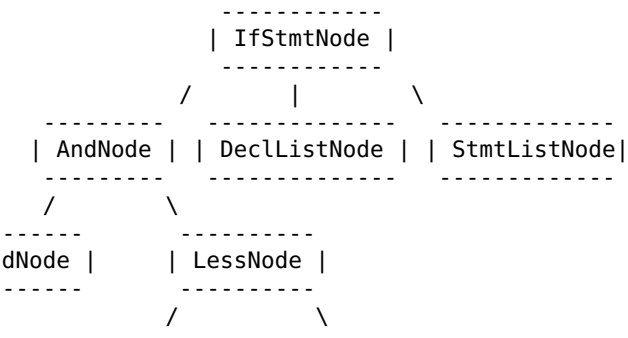
the left child. That call will generate code to evaluate the left operand, jumping to the "TrueLabel" that we pass if its value is true, and jumping to the "FalseLabel" that we pass if its value is false.

- So what labels should be passed?
  - If the left operand is false, then the value of the whole expression is false, so pass the given "FalseLabel" as the "FalseLabel" in the recursive call.
  - If the left operand is true, then we must evaluate the right operand, so pass a *new* label as the "TrueLabel" in the recursive call.
- After the call to the left child's genJumpCode method, call genLabel to generate the new label.
- Finally, call the genJumpCode method of the right child.
- What labels should be passed as the arguments to the right child's genJumpCode method?
  - This expression will only be evaluated if the left operand evaluated to true; in that case, the value of this expression (the right operand) is the value of the *whole* expression. Therefore, for this recursive call, pass the original "TrueLabel" and "FalseLabel".

The AndNode's genJumpCode method would be:

```
public void genJumpCode(String trueLab, String falseLab) {
    String newLab = nextLabel();

    myExp1.genJumpCode(newLab, falseLab);
    genLabel(newLab);
    myExp2;genJumpCode(trueLab, falseLab);
}
```

Example: Consider the code that would be generated for the statement: `if (a && b>0) { ... }`, represented by the AST:



```
                ...         ...
```

The IfStmtNode's codeGen method would create two labels, TrueLab and DoneLab, and would call the AndNode's genJumpCode method, passing those labels as the arguments. The AndNode's genJumpCode method would create one new label (NewLab) and then would call the genJumpCode method of its left child (the IdNode), passing NewLab and DoneLab as the arguments. It would then generate the NewLab label, and then would call its right child's genJumpCode method, passing TrueLab and DoneLab. The code generated for the whole condition would look like this:

```
          Generated Code                                  Generated By
          --------------                                  ------------

             -- code to load the value of a into T0          IdNode
             jump to DoneLab if T0 == FALSE                 IdNode
             jump to NewLab                                 IdNode
  NewLab:                                                   AndNode
             -- code to push the value of b                 LessNode's child
             -- code to push the iteral 0                   LessNode's child
             pop into T1                                    LessNode
             pop into T0                                    LessNode
             jump to TrueLab if T0 < T1                     LessNode
             jump to DoneLab                                LessNode
```

(Of course, the actual label would be something like `L3`, not `NewLab`.) After calling the AndNode's genJumpCode method, the IfStmtNode's codeGen method would call genLabel to print TrueLab, then would call its StmtList child's codeGen method to generate code for the list of statements. Finally, it would call genLabel to print DoneLab. So the code generated for this if statement would be like this:

```
             -- code to load the value of a into T0
             jump to DoneLab if T0 == FALSE
             jump to NewLab
  NewLab:
             -- code to push the value of b
             -- code to push the iteral 0
             pop into T1
             pop into T0
             jump to TrueLab if T0 < T1
             jump to DoneLab
  TrueLab:
             -- code for the list of statements
  DoneLab:
```

TEST YOURSELF #3

Question 1: What is the form of the code

generated by an OrNode's genJumpCode method?

Question 2: What is the form of the code generated by a NotNode's genJumpCode method?

solution

How does the code generated for an AndNode using the control-flow method compare to the code generated using the numeric method? Here are outlines of the code generated in each case:

```
    Numeric Code                                  Control-Flow Code
    ------------                                  -----------------

    -- code to evaluate left                         -- code to evaluate left
    -- operand, leaving the                          -- operand, including jumps
    -- value on the stack                            -- to NewLab and FalseLab
    pop into T0
    goto TrueLab if T0 == TRUE                 newLab:
    push FALSE
    goto DoneLab
 TrueLab:
    -- code to evaluate right                        -- code to evaluate right
    -- operand, leaving the                          -- operand, including
    -- value on the stack                            -- jumps to TrueLab and
                                                     -- FalseLab
 DoneLab:
```

Note that the numeric code includes 6 instructions (shown in bold) in addition to the ones generated by the codeGen methods of the two children, while in the control-flow case, *no* instructions are generated by the AndNode itself (just a label); the instructions are all generated by the genJumpCode methods of its two children. Those children could represent names, boolean literals, comparisons or logical expressions. We have already seen that in the first three cases, better code is generated using the control-flow approach. Now we see that for logical expressions (at least for AndNodes) fewer instructions are generated using the control-flow approach than using the numeric approach.

TEST YOURSELF #4

Compare the code generated by the two approaches for an OrNode and for a NotNode.

solution

Finally, let's compare the code generated by the numeric and control-flow approaches for an if-then statement. Here are outlines of the two different versions of the code that would be generated:

```
    Numeric Code                                  Control-Flow Code
    ------------                                  -----------------

    -- code for condition, leaving                -- code for condition,
    -- value on the stack                         -- including jumps to
                                                  -- trueLab and falseLab
    pop into T0
    goto falseLab if T0 == FALSE               trueLab:
    -- code for "then" stmts                      -- code for "then" stmts
    goto doneLab                                  goto doneLab
falseLab:                                      falseLab:
    -- code for "else" stmts                      -- code for "else" stmts
doneLab:                                       doneLab:
```

Note that much of the code is the same for the two methods; the code generated to evaluate the condition will be different, and the numeric method has three extra instructions: a pop, followed by a conditional goto (those instructions are shown in bold font). So as long as the code generated for the condition is no worse for the control-flow method than for the numeric method, the control-flow method is better (both in terms of the number of instructions generated, and in terms of the number of instructions that will be executed).

But we have already looked at the code generated for all the different kinds of conditions, for each of the two approaches, and in fact the control-flow method was never worse, and was sometimes better. Thus, the control-flow method is the winner, in terms of generating less and more efficient code!

- Overview
- Peephole Optimization
  - Test Yourself #1
- Moving Loop-Invariant Computations
  - Test Yourself #2
- Strength Reduction in *for* Loops
  - Test Yourself #3
  - Test Yourself #4
- Copy Propagation

Overview

The goal of optimization is to produce better code (fewer instructions, and, more importantly, code that runs faster). However, it is important not to change the behavior of the program (what it computes)!

We will look at the following ways to improve a program:

1. **Peephole Optimization**. This is done ***after*** code generation. It involves finding opportunities to improve the generated code by making small, local changes.
2. **Moving Loop-Invariant Computations**. This is done ***before*** code generation. It involves finding computations inside loops that can be moved outside, thus speeding up the execution time of the loop.
3. **Strength-Reduction in *for* Loops**. This is done ***before*** code generation. It involves replacing multiplications inside loops with additions. If it takes longer to execute a multiplication than an addition, then this speeds up the code.
4. **Copy Propagation**. This is done ***before*** code generation. It involves replacing the use of a variable with a literal or another variable. Copy propagation can sometimes uncover more opportunities for moving loop-invariant computations. It may also make it possible to remove some

assignments from the program, thus making the code smaller and faster.

Peephole Optimization

The idea behind peephole optimization is to examine the code "through a small window," looking for special cases that can be improved. Below are some common optimizations that can be performed this way. Note that in all cases that involve removing an instruction, it is assumed that that instruction is not the target of a branch.

1. Remove a redundant load (fewer instructions generated, and fewer executed):

2. Remove a redundant push/pop (fewer instructions generated, and fewer executed):

3. Replace a jump to a jump (same number of instructions generated, but fewer executed):

4. Remove a jump to the next instruction (fewer instructions generated, and fewer executed):

5. Replace a jump around jump (fewer

instructions generated; possibly fewer executed):

```
   if T0 == 0 goto L1
   goto L2
L1:   ...
```

after peephole optimization →

```
   if T0 != 0 goto L2
L1:   ...
```

6. Remove useless operations (fewer instructions generated and fewer executed):

```
add T0,T0, 0
mul T0,T0, 1
```

after peephole optimization → nothing! (Adding 0 or multiplying by 1 has no effect: these instructions are useless)

7. Reduction in strength: don't use a slow, general-purpose instruction where a fast, special-purpose instruction will do (same number of instructions, but faster):

```
mul, T0, T0, 2
```

after peephole optimization → shift-left T0

```
add, T0, T0, 1
```

→ inc T0

Note that doing one optimization may enable another: for example:

```
load  Tx, M
add   Tx, 0
store Tx, M
```

after round 1 →

```
load  Tx, M
store Tx, M
```

after round 2 →

```
load  Tx, M
```

TEST YOURSELF #1

Consider the following program:

```
public class Opt {

   public static void main() {
       int a;
       int b;

       if (true) {
         if (true) {
            b = 0;
         }
         else {
```

```
            b = 1;
          }
          return;
        }
        a = 1;
        b = a;
    }
}
```

Question 1: The code generated for this program contains opportunities for the first two kinds of peephole optimization (removing a redundant load, and replacing a jump to a jump). Can you explain how those opportunities arise just by looking at the source code?

Question 2: Below is the generated code. Verify your answer to question 1 by finding the opportunities for the two kinds of optimization. What other opportunity for removing redundant code is common in this example?

```
.text
        .globl main
main:           # FUNCTION ENTRY
        sw    $ra, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        sw    $fp, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        addu  $fp, $sp, 8
        subu  $sp, $sp, 8
        # STATEMENTS
        # if-then
        li    $t0, 1
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
        beq   $t0, 0, _L0
        # if-then-else
        li    $t0, 1
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
        beq   $t0, 0, _L1
        li    $t0, 0
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
        sw    $t0, -12($fp)
        b     _L2
_L1:
        li    $t0, 1
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
```

```
        sw    $t0, -12($fp)
_L2:
        # return
        b     main_Exit
_L0:
        li    $t0, 1
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
        sw    $t0, -8($fp)
        lw    $t0, -8($fp)
        sw    $t0, 0($sp)       #PUSH
        subu  $sp, $sp, 4
        lw    $t0, 4($sp)       #POP
        addu  $sp, $sp, 4
        sw    $t0, -12($fp)
                #FUNCTION EXIT
main_Exit:
        lw    $ra, 0($fp)
        move  $sp, $fp          #restore SP
        lw    $fp, -4($fp)      #restore FP
        jr    $ra               #return
```

Optimization #1: Loop-Invariant Code Motion

The ideas behind this optimization are:

- For greatest gain, optimize "*hot spots*" i.e., inner loops.
- An expression is "loop invariant" if the *same* value is computed for that expression on every iteration of the loop.
- Instead of computing the same value over and over, compute the value once outside the loop and reuse it.

Example:

```
  for (i=0; i<100; i++) {
      for (j=0; j<100; j++) {
          for (k=0; k<100; k++) {
              A[i][j][k] = i*j*k
          }
      }
  }
```

In this example, i*j is invariant with respect to the inner loop. But there are *more* loop-invariant expressions; to find them, we need to look at a lower-level version of this code. If we assume the following:

- A is a 3D array

- each element requires 4 bytes
- elements are stored in the current activation record in row-major order (note: in Java, arrays are allocated from the heap, not stored on the stack; however, in other languages they may be stored on the stack)

then the code for A[i][j][k] = ... involves computing the *address* of A[i][j][k] (i.e., where to store the value of the right-hand-side expression). That computation looks something like:

address = FP - <offset of A> + (i*10,000*4)+(j*100*4)+(k*4)

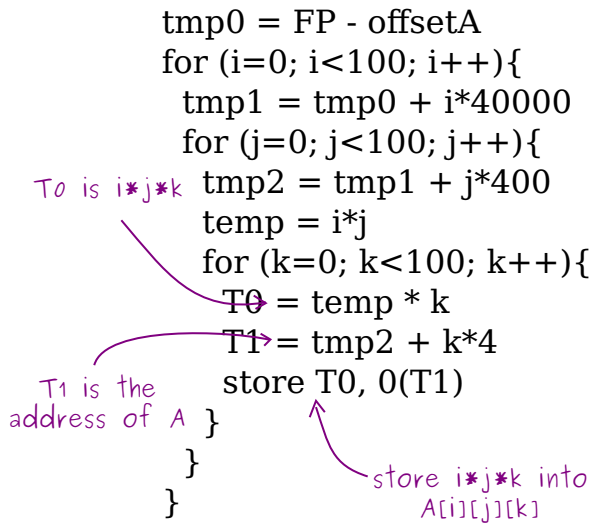
So the code for the inner loop is actually something like:

T0 = i*j*k
T1 = FP - <offset of A> + i*40000 + j*400 + k*4
Store T0, 0(T1)

And we have the following loop-invariant expressions:

invariant to i loop: FP - <offset of A>
invariant to j loop: i*40000
invariant to k loop: i*j, j*400

We can move the computations of the loop-invariant expressions out of their loops, assigning the values of those expressions to new temporaries, and then using the temporaries in place of the expressions. When we do that for the example above, we get:



Here is a comparison of the original code and the optimized code (the number of instructions performed in the innermost loop, which is executed 1,000,000 times):

Original Code	**New Code**
5 multiplications (3 for lvalue, 2 for rvalue)	2 multiplications (1 for lvalue, 1 for rvalue)
1 subtraction; 3 additions (for lvalue)	1 addition (for lvalue)
1 indexed store	1 indexed store

Questions:

1. How do we recognize loop-invariant expressions?
2. When and where do we move the computations of those expressions?

Answers:

1. An expression is *invariant* with respect to a loop if for every operand, one of the following holds:
    a. It is a literal, or
    b. It is a variable that gets its value *only* from outside the loop.

2. To answer question 2, we need to

consider *safety* and *profitability*.

Safety

If evaluating the expression might cause an *error*, then there is a possible problem if the expression *might not* be executed in the original, unoptimized code. For example:

```
b = a;
while (a != 0){
  x = 1/b;
  a--;
}
```

possible divide by zero if moved out of the loop

What about preserving the *order* of events? e.g. if the unoptimized code performed output then had a runtime error, is it valid for the optimized code to simply have a runtime error? Also note that changing the order of floating-point computations may change the result, due to differing precisions.

Profitability

If the computation might *not* execute in the original program, moving the computation might actually slow the program down!

Moving a computation is both safe and profitable if one of the following holds:

1. It can be determined that the loop will execute at least once and the code is guaranteed to execute if the loop does:
   - it isn't inside any condition, or
   - it is on *all* paths through the loop (e.g., it occurs in both branches of an if-then-else).
2. The expression is in (a non short-circuited) part of the loop test / loop bounds, e.g.:

   ```
   while (x < i + j * 100) //
   j*100 will always be evaluated
   ```

TEST YOURSELF #2

What are some examples of loops for which the compiler can be sure that the loop will execute at least once?

solution

Optimization #2: Strength reduction in for-loops

The basic idea here is to take advantage of patterns in for-loops to replace expensive operations, like multiplications, with cheaper ones, like additions.

The particular pattern that we will handle takes the general form of a loop where:

1. `L` is the loop index
2. `B` is the beginning value of the loop
3. `E` is the end value of the loop
4. The body of the loop contains a right-hand-side expression of the form `L * M + C`. We call this the *induction expression*.
5. The factors of the induction expression, `M` and `C`, must be constant with respect to the loop.

These rules define a sort-of "template" of the following form* :

```
for L from B to E do {
        ⋮
   ... = L * M + C
        ⋮
}
```

Consider the sequences of values for `L` and for the induction expression:

Iteration #	L	L * M + C
1	B	B * M + C

2	B + 1	(B + 1) * M + C = B * M + M + C
3	B + 1 + 1	(B + 1 + 1) * M + C = B * M + M + M + C

Note that in each case, the part of the induction expression highlighted in orange is the same as the value of the whole expression on the previous iteration, and the non-highlighted part each time is always `+ M`. In other words, each time around the loop, the induction expression increases by adding `M`, a *constant* value! So we can avoid doing the multiplication each time around the loop by:

- computing `B * M + C` once before the loop,
- storing that value in a temporary,
- using the temporary instead of the expression inside the loop, and
- incrementing the temporary by `M` at the end of the loop.

Here is the transformed loop:

```
ind = B * M + C //Initialize temp
to first value of expression
for L from B to E do {
        ⋮
   ... = ind //Use ind instead of
recalculating expression
        ⋮
   ind = ind + M //Increment ind at
the end of the loop by M
}
```

Note that instead of doing a multiplication and an addition each time around the loop, we now do just one addition each time. Although in this example we've *removed* a multiplication, in general we are replacing a multiplication with an addition (that is why this optimization is called *reduction* in strength). Although this pattern may

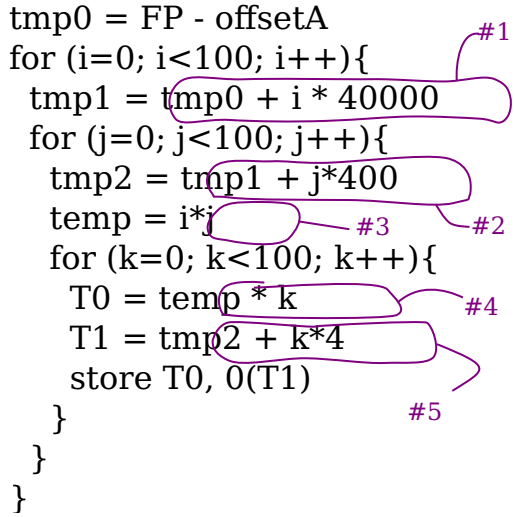
seem restrictive, in practice many loops fit into this template, especially since we allow `M` or C to be absent. In particular, if there were no `C`, the original induction expression would be: `L * M`, and that would be replaced inside the loop by: `ind = ind + M`; an addition replaces a multiplication.

TEST YOURSELF #3

Some languages actually have for-loops with the syntax used above (`for i from low to high do ...`), but other languages (including Java) do not use that syntax. Must a Java compiler give up on performing this optimization, or might it be able to recognize opportunities in some cases?

solution

As mentioned above, many loops naturally fit the template for strength reduction that we defined above. Now let's see how to apply this optimization to the example code we used to illustrate moving loop-invariant computations out of the loop. Below is the code we had after moving the loop-invariant computations. Each induction expression is circled and identified by a number:



Original Expression	Loop Index (L)	Multiply Term (M)	Additio Term (C)
#1: tmp0 + i * 40000	i	40000	tmp0
#2: tmp1 + j * 400	j	400	tmp1
#3: i * j	j	i	0
#4: temp * k	k	temp	0
#5: tmp2 + k * 4	k	4	tmp2

After performing the reduction in strength optimizations:

```
tmp0 = FP - offsetA
ind1 = tmp0  <-- ind1 = 0*40000+tmp0
for (i=0; i<100; i++){
  tmp1 = ind1
  ind2 = tmp1    <-- ind2 = 0*400+tmp1
  ind3 = 0     <-- ind3 = 0*i+0
  for (j=0; j<100; j++){
    tmp2 = ind2
    temp = ind3
    ind4 = 0      <-- ind4 = 0*temp+0
    ind5 = tmp2      <-- ind5 = 0*4+tmp2
    for (k=0; k<100; k++){
      T0 = ind4
      T1 = ind5
      store T0, 0(T1)
      ind4 = ind4 + temp
      ind5 = ind5 + 4
    }
    ind2 = ind2 + 400
    ind3 = ind3 + i
  }
  ind1 = ind1 + 40000
}
```

In the original code, the innermost loop (executed 1,000,000 times) had two multiplications and one subtraction. In the optimized code, the inner loop has no multiplications, one subtraction, and one

addition. (Similarly, the middle loop went from two multiplications and one subtraction to no multiplications, one subtraction, and one addition; the outer loop went from one multiplication and one subtraction to no multiplications and one subtraction.) On the other hand, we have added a number of assignments; for example, the inner loop had just two assignments, and now it has four. We'll deal with that in the next section using copy propagation

---

TEST YOURSELF #4

Suppose that the index variable is incremented by something other than one each time around the loop. For example, consider a loop of the form:

```
for (i=low; i<=high; i+=2) ...
```

Can strength reduction still be performed? If yes, what changes must be made to the proposed algorithm?

solution

Optimization #3: Copy propagation

Statements of the form `x = y` (call this definition d) are called *copy statements*. For every use u of variable `x` reached by definition d such that:

1. no other definition of `x` reaches u, and
2. `y` can't change between d and u

we can replace the use of `x` at u with a use of `y`.

Examples:

```
x = y
a = x+z
```

x can be replaced with y

```
x = y
if (...) x = 2
a = x+z
```

x cannot be replaced with y; violates condition 1

```
x = y
if (...) y = 3
a = x+z
```

x cannot be replaced with y; violates condition 2

Question: Why is this a useful transformation?

Answers:

- If <u>all</u> uses of `x` reached by d are replaced, then definition d is *useless*, and can be removed.

- Even if the definition cannot be removed, copy propagation can lead to improved code:
  1. If the definition is actually of the form: `x = literal`, then copy propagation can create opportunities for better code:

     ```
     x = 5;
     a = b + x;
     ```

     after copy propagation

     ```
     x = 5;
     a = b + 5;
     ```

     For a machine like the MIPS, there are fast instructions that can be used when one of the operands is an "immediate" (literal) value. These instructions can be used for operation `a = b + 5`, since one of the operands is a literal, but not for `a = b + x`. The improvemnt is even more striking because MIPS doesn't allow arithmetic operands to be memory locations, so to generate assembly for statements like `a = b + x`, it would be necessary to load the values for both `b` and `x` into registers, which would require additional `load` instructions*

     Furthermore, this kind of copy

propagation can lead to opportunities for *constant folding*: evaluating, at compile time, an expression that involves only literals. For example:

2. Sometimes copy propagation can be combined with moving loop-invariant computations out of the loop, to lead to a better overall optimization. For example:

```
while (...) {
   x = a * b; // loop-inv
   y = x * c;
   ...
}
```

Move "`a * b`" out of the loop:

```
tmp1 = a * b;
while (...) {
   x = tmp1;
   y = x * c;
   ...
}
```

Note that at this point, even if `c` is not modified in the loop, we cannot move "`x * c`" out of the loop, because `x` gets its value inside the loop. However, after we do copy propagation:

```
tmp1 = a * b;
while (...) {
   x = tmp1;
   y = tmp1 * c;
   ...
}
```

"`tmp1 * c`" can also be moved out of the loop:

```
tmp1 = a * b;
tmp2 = tmp1 * c;
while (...) {
   x = tmp1;
   y = tmp2;
   ...
}
```

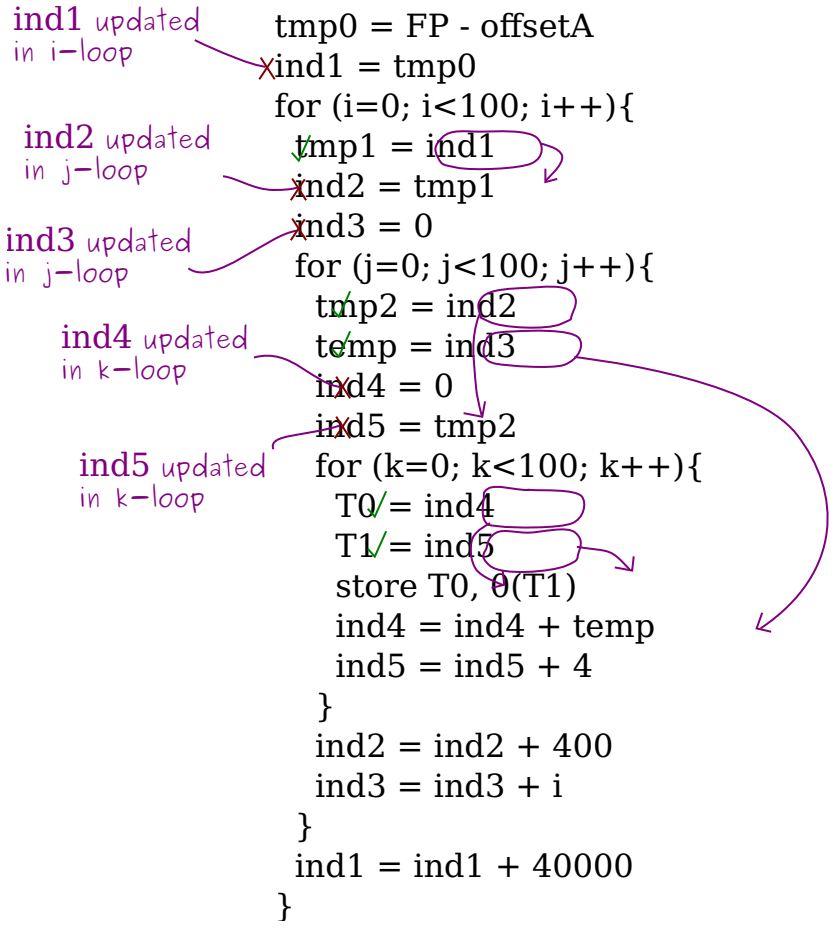
Given a definition d that is a copy statement: `x = y`, and a use u of `x`, we

must determine whether the two important properties hold that permit the use of `x` to be replaced with `y`.

The first property (use u is reached only by definition d) is best solved using the standard "reaching-definitions" dataflow-analysis problem, which computes, for each definition of a variable `x`, all of the uses of `x` that might be reached by that definition. Note that this property can also be determined by doing a backward depth-first or breadth-first search in the control-flow graph, starting at use u, and terminating a branch of the search when a definition of `x` is reached. If definition d is the *only* definition encountered in the search, then it is the only one that reaches use u. (This technique will, in general, be less efficient than doing reaching-definitions analysis.)

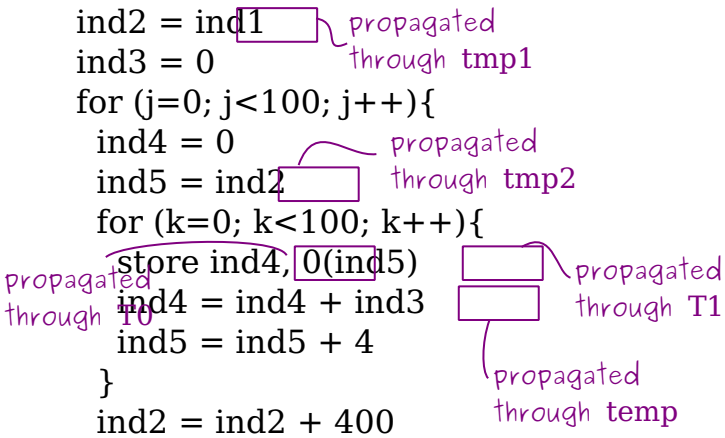
The second property (that variable `y` cannot change its value between definition d and use u), can also be verified using dataflow analysis, or using a backwards search in the control-flow graph starting at u, and quitting at d. If no definition of `y` is encountered during the search, then its value cannot change, and the copy propagation can be performed. Note that when `y` is a literal, property (2) is always satisfied.

Below is our running example (after doing reduction in strength). Each copy statements either has a red X next to it (if it can't be propagated) or a green check (if it can be propagated). In this particular example, each variable x that is defined in a copy statement reaches only *one* use. Comments indicate which of them cannot be propagated (because of a violation of property (1) -- in this example there are no instances where property (2) is violated).



Here's the code after propagating the copies that are legal, and removing the copy statements that become dead. Note that we are able to remove 5 copy statements, including 2 from the innermost loop.

```
tmp0 = FP - offsetA
ind1 = tmp0
for (i=0; i<100; i++){
  ind2 = ind1
  ind3 = 0
  for (j=0; j<100; j++){
    ind4 = 0
    ind5 = ind2
    for (k=0; k<100; k++){
      store ind4, 0(ind5)
      ind4 = ind4 + ind3
      ind5 = ind5 + 4
    }
    ind2 = ind2 + 400
    ind3 = ind3 + i
  }
  ind1 = ind1 + 40000
}
```



Comparing this code with the original code, we see that, in the inner loop (which is executed 1,000,000 times) we originally had 5 multiplications, 3 additions/subtractions, and 1 indexed store. We now have no multiplications and just 2 additions/subtractions. We have added 2 additions/subtractions and 2 copy statements to the middle loop (which executes 10,000 times) and 1 addition/subtraction and 1 copy statement to the outer loop (which executes 100 times), but overall this should be a win!