Selected Topics in Imperative Languages

Overview

Grammars
- Context-Free
- Ambiguous

Parameter-Passing
  By value
  By reference
  By value-result
  By name

Notion of Scope
  Static
  Dynamic

Reading: Chapters 2.4-2.6 and 5.1-5.4 in Sethi
Homeworks 1 and 2 (with solutions) are on the Web.

Grammars

Q: How to describe a syntax of a programming language?
A: Use a <u>formal grammar</u>.

Example:

```
  (1) <S> -> <NP> <VP>
"A sentence is a NounPhrase followed by a Verb
Phrase."
  (2) <NP> -> <Noun> | <Adj><Noun>
"A Noun Phrase is either a noun of an adjective
followed by a noun."
  (3) <VP> -> <Verb> | <Verb><NP>
"A Verb Phrase is either a verb or a verb
followed by a noun phrase."
  (4) <Noun> -> Tarzan | Jane
  (5) <Verb> -> go | like | hit
  (6) <Adj>  -> strong | pretty
```


Sample Derivation 1

```
<S> -> <NP>   <VP>              by (1)
    -> <Noun> <VP>              by (2)
    -> <Noun> <Verb>            by (3)
    -> Tarzan <Verb>            by (4)
    -> Tarzan go                by (5)
```

Parse Tree:

```
        <S>
       /   \
    <NP>   <VP>
     |       |
   <Noun>  <Verb>
     |       |
   Tarzan    go
```

• Note that leaves of the trees contain terminals, whereas other nodes in the tree are nonterminals.
• <S> is a <u>starting nonterminal</u>.

Sample Derivation 2

```
<S> -> <NP>   <VP>                     by (1)
  -> <NP> <Verb> <NP>                  by (3)
  -> <Adj> <Noun> <Verb> <NP>          by (2)
  -> <Adj> <Noun> <Verb> <Adj> <Noun>  by (2)
  -> pretty Jane hit strong Tarzan     by (4)-(6)
```

Parse Tree:

```
                <S>
              /     \
             /       \
          <NP>       <VP>
         /   \       /   \
        /     \     /     \
     <Adj>  <Noun> <Verb> <NP>
       |      |      |    /  \
       |      |      |   /    \
   pretty   Jane   hit  strong Tarzan
```

Recursive Grammars

```
Grammar:  <NP> -> <Noun> | <Adj> <NP>
Sample derivation:
  <NP> -> <Adj> <NP>
       -> <Adj> <Adj> <NP>
       -> <Adj> <Adj> <Adj> <NP>
       -> <Adj> <Adj> <Adj> <Noun>
       -> strong strong pretty Tarzan
The Parse Tree:
          <NP>
         /    \
    <Adj>     <NP>
     |       /    \
     |   <Adj>     <NP>
     |     |      /    \
     |     |  <Adj>    <NP>
     |     |     |      |
     |     |     |    <Noun>
     |     |     |      |
  strong strong pretty Tarzan
```


Some Definitions

- A *grammar* is a set of rules for generating *strings* in a *language*.
- Given a set of symbols (terminals and non-terminals), a *string* over the set is a finite sequence of zero or more symbols from the set.
- The number of symbols in the sequence is the *length* of the string.
- An *empty* string is a string of length zero.

Context-Free Grammars

A Context-Free grammar has four parts:

1. A set of tokens or terminals that are the atomic symbols in the language
   E.g. `Tarzan, Jane, hit, pretty`.

2. A set of nonterminals; these are the variables representing constructs in the language
   E.g. <NP>, <VP>, <Adj>.

3. A set of rules called productions for identifying the components of a construct. Each production has a nonterminal as its left side, the symbol $->$ (or sometimes ::=), and a string over the sets of terminals and nonterminals as its right side.

4. A nonterminal chosen as a starting nonterminal
   E.g. <S>.

More Examples

So, grammars are often used to describe set of strings - a *language*.

Ex: `S -> b | bS`

This rule describes all strings of b's:

```
S -> b
--------
S -> bS
  -> bb
--------
S -> bS
  -> bbS
  -> bbb
--------
etc.
```

We will use the following shorthand when describing languages:

- a^* - zero or more a's
- a^+ - one or more a's
- a^n - sequence of n a's.

So, the above grammar generates a language b^+.

Examples (Cont'd)

Another example:

```
S -> b | bSb
```

This rule describes all strings of `b`'s of <u>odd</u> length:

```
S -> b
----------
S -> bSb
  -> bbb
----------
S -> bSb
  -> bbSbb
  -> bbbbb
----------
etc.
```

We can represent the language generated by this grammar as $(bb)^+b$

More examples

```
Grammar:  S -> A B      (1)
          A -> a | aA   (2)
          B -> b | bB   (3)
Derivation:
          S -> AB       by (1)
            -> AbB      by (3)
            -> AbbB     by (3)
            -> Abbb     by (3)
            -> aAbbb    by (2)
            -> aabbb    by (2)
```

This describes strings consisting of any number of `a`'s followed by any number of `b`'s, or `a*b*`.

Ambiguous Grammars

- A grammar is <u>ambiguous</u> if it generates a string that has more than one parse tree.
- To prove that a grammar is ambiguous, need to find one string and show two different derivations for it.

```
Ex.  S -> AB
     A -> a | ac
     B -> b | cb

String acb has two parse trees:

     S             S
    / \           / \
   A   B         A   B
  / \  |         |  / \
  a c  b         a  c  b
```


Grammar for Arithmetic Expressions

```
<Exp> -> <Exp> + <Exp>
<Exp> -> <Exp> * <Exp>
<Exp> -> 1 | 2 | 3 | 4 | ...
```

This grammar is ambiguous. String `1 + 2 * 3` has <u>two</u> parse trees:

```
     <Exp>                          <Exp>
    /  |    \                     /    |  \
<Exp>  |     <Exp>          <Exp>      |  <Exp>
  |    |   /  |   \        /  |   \    |    |
  |    | <Exp> | <Exp>  <Exp> | <Exp> |    |
  |    |   |  |   |       |   |   |    |    |
  1    +   2  *   3       1   +   2    *    3
(corresponds to 1+(2*3))
                       (corresponds to (1+2)*3)
```

Grammars for Arithmetic Expressions (Cont'd)

By introducing brackets, many strings in the language can be disambiguated. So, add the following rule to the language:

$<$ `Exp` $> - >$ ($<$ `Exp` $>$)

In the new language, strings `(1 + 2) * 3` and `1 + (2 * 3)` are different. Each has only one tree.

Is this grammar still ambiguous? Yes!

- We can still generate strings `1 + 2 * 3` different ways.

Solution: insert precedence rules.

```
<Exp>     -> <Exp> + <Exp> | <Term>
<Term>    -> <Term> * <Term> | <Functor>
<Functor> -> number | (<Exp>)
```

Is this grammar still ambiguous?

Yes! What about `1 + 2 + 3`?

```
      <Exp>                        <Exp>
    /  |    \                    /    |  \
<Exp>  |     <Exp>          <Exp>     |  <Exp>
  |    |   /   |   \      /   |   \   |    |
  |    | <Exp> | <Exp>  <Exp> | <Exp> |    |
  |    |   |   |   |      |   |   |   |    |
  1    +   2   +   3      1   +   2   +    3
```

Final attempt. Insert associativity. + and * associate to the left.

```
<Exp>     -> <Exp> + <Term> | <Term>
<Term>    -> <Term> * <Functor> | <Functor>
<Functor> -> number | (<Exp>)
```

This grammar seems to be OK.

Examples: Given Language, Find Grammar

Ex. Give unambiguous grammar for language $\{a^*\}$.

- We want strings like ϵ, a, aa, aaa, etc.
- So, each time we want to generate one a.
- Grammar is S $- >$ aS $\mid \epsilon$

Ex: Give unambiguous grammar for language $\{a^n b^n \mid n \geq 0$ and n is even $\}$.

- We want strings like ϵ, aabb, aaaabbbb, aaaaaabbbbbb.
- So, each time we want two a's and two b's generated.
- Grammar is S $- >$ aaSbb $\mid \epsilon$

More Examples

Ex. Give unambiguous grammar for language $\{aba^*\}$

- We want strings like ab, aba, abaa, abaaa, etc.
- So, need to generate ab, followed by a grammar generating a^*.
- Our thinking: generate ab and then move to another nonterminal. Use it to generate a^*.
- Grammar is

```
S -> abP
P -> aP | e
```

Grammars: Summary

- Used to formally specify syntax of languages
- Can be English, Prolog, etc.
- Context-free grammars
- Problem of parsing is solved:
  - Automatic tools for generating parsing code given context-free grammar (Yacc)
  - Do lots of parsing in 488 (Compiler Construction)
- Can determine if a string (a language) is generated by a grammar
- Can find a grammar given a language
- Grammars are ambiguous if a string can be generated two different ways.

Imperative Languages - Motivation and Overview

- In these languages, we specify *how* to solve the problem, giving step by step instructions.
- Notions of modularity, abstraction, information-hiding.
- Based on John van Neumann's notion of computing - program is to modify some memory.
- Central are ideas of state, of variables, of assignment.

We will look at some important notions in imperative languages.

Questions of Interest

1. When is there a mapping between variable name and its declarations, i.e., which declaration applies to a given occurrence of `x`?

2. When is there a mapping between declaration and storage location, i.e., which location does the name in declaration denote?

3. When is there a mapping from locations to values? Does occurrence of a variable deal with value or location? E.g., `x := x + 1` changes value, not location.

Procedures and Functions

- *Procedures* do some action.
- *Functions* return some value.

Pure functions have no side effects (as in Scheme or ML):

```
function square(x:int) :  int;
begin
  return x*x;
end
```

Pure procedures do not return anything:

```
procedure write(x:int);
begin
  write ("I am about to write a variable ");
  write (x);
end;
```

Parameter-Passing Methods

Formal parameters in procedure definition are just place holders, replaced by actuals during procedure call.
Parameter passing - matching of actuals with formals when procedure call occurs. But... what does `A[i]` mean: a name? a value? an address?

Possible interpretations:
- call by value: pass value of `A[i]`
- call by reference: pass location of `A[i]`
- call by name: pass string "A[i]"
- call by value-result: copy values of actuals into formals on entrance, copy values of formals into actuals on exit.

Call by Value

Evaluate the expression and then pass its value to the procedure.
Example:

```
procedure swap(x,y : T);
var z : T;
begin
  z := x; x := y; y := z;
end;
```

When we call this procedure:

```
a := 3;
b := 4;
swap(a,b);
```

the following occurs:

```
x := a;
y := b;
z := x; x := y; y := z;
```

So, values of `a` and `b` do not change.

Call by Value (Cont'd)

Expressions (e.g., all parameters that do not have addresses, *lvalues*) are always passed by value.

Disadvantages:
- changes made to the parameter during the procedure call do not get reflected when the procedure returns.
- have to copy the object (OK for small objects, bad for big).

So, what do we do if we want to modify parameters? Use pointers (as in C), return some parameters, or use call by reference (in C++, Turing, Pascal)

Call by Reference

A formal parameter becomes a synonym for the location of the actual parameter. In contemporary languages, all parameters passed by reference have to have a location. In early Fortran, could swap constants which was very confusing.

In Pascal, call by reference is marked by keyword `var`:

```
procedure swap (var x,y : integer);
var z : integer;
begin
  z := x; x := y; y := z;
end;
```

So, in the call

```
i := 2;
A[i] := 3;
swap(i, A[i]);
```

values of `i` and `A[i]` do get swapped.

Call by Reference (Cont'd)

The language implementation does the following: make location of `x` same as that of `i`, make location of `y` same as `A[i]` and then call the procedure.

Advantages:
- do not have to make a copy of the object
- do not have to dereference pointers

Disadvantages:
- can incidently modify a parameter.

Best if the language allows both call by value and by reference.

Call by Value-Result

Another name for this is *copy-in/copy-out*.
1. Copy-in phase. Compute values and locations of actual parameters. Values are assigned to the corresponding formals, and locations are stored.
2. Copy-out phase. After procedure body is executed, the final values are copied back to computed locations.

Call by value is used in Ada which has either value-result or reference parameters. Thus, all formals have to be specified as in, out and in-out parameters.

Call by value works the same as by reference unless aliasing is used, i.e., several names for the same location. Can change value of variable directly using assignment or through a formal.

Call by Value-Result (Cont'd)

Typical example:

```
procedure f (x,y : integer);
begin
  x := x+1;
  b := x;
  y := y-1;
  write (x,y);
end;
begin (* main *)
  a = 1; b := 2;
  f (a,b);
  write (a,b);
end;
```

Output by reference:

Output by value-result:

Call by Name

Pass a name string, resolve it only when it is necessary. In functional languages, called "lazy" evaluation.
Example:

```
procedure print (a : integer);
begin
  i := 3; write (a);
end;
begin (* main *)
  A[1] := 1;
  A[2] := 2;
  A[3] := 3;
  i := 2;
  print(A[i]);
end;
```

This prints "2" under first three mechanisms. However, by name this prints ...

Notion of Scope

Scope rules determine which declaration of a variable (or a function) applies to a name. Two main types of scope - lexical (static) and dynamic. Scopes can be nested.

Static scoping - names are resolved during compile time. Variables come from declaration scope. (Scheme, ML, C, C++). General rule - consistent renaming of formal parameters should not change the function behavior.

- *Dynamic* scoping - names are resolved at run-time and come from calling scope. (Lisp)

Notion of Scope(Cont'd)

Example:

```
procedure L
  var n:char;
  procedure W
    begin writeln(n); end
  procedure D
    var n: char
    begin
      n := 'D'; W;
    end;
  begin
    n := 'L';
    W;  D;
  end;
```

Under static scoping:
Under dynamic scoping:

Interesting Case - Static Scoping with Macros

Macro processor does the following:
1. Actual parameters are textually substituted for the formals.
2. The resulting procedure body is textually substituted for the call.

Example:

```
var n :char = 'A'
#define WRITE {writeln(n);}
procedure D;
begin
  var n:char = 'D';
  WRITE;
end;
```

This prints 'D'. So, we get dynamic scoping!

Call by Name

Want lexical scoping. Rename locals:
1. Actual parameters are substituted for formals. Locals are renamed if there are conflicts.
2. Resulting procedure body is substituted for the call. Rename locals at point of call if there are conflicts between nonlocals in proc. and locals at point of call.

```
procedure P(x)
begin
  i :  integer;
  i := i+n; x := x+n;
end;
begin (* main *)
  n, i :  integer;
  P(A[i]);
end;
```

Here, replace `i` by `j` in body of procedure, replace `n` by `m` in calling procedure.

Interesting Case - Static Scoping with Call by Name

```
program P(output);
var A : array[1..4] of integer;
    I,J : integer;
procedure P(X,Y,Z:integer);
  begin
    I := I+1; J := J+2; X := X*Z;
    Y := 0;
  end;
procedure Q;
  var J,K : integer;
  begin
    J := 2; K := 3;
    R(A[I], A[J], J+K);
  end;
begin (* main *)
  for J := 1 to 4 do A[J] := 3;
  J := 1; I := 2;
  Q;
  writeln (A[1], A[2], A[3], A[4]);
end;
```

What does this program print?

food for slide eater

food for slide eater

food for slide eater