Subprograms

1

Subprograms

- The term *subprogram* will be used to mean either a *procedure* or a *function*.
- We have already addressed subprograms and issues of scope within the scanner, parser, and identifier table, so most of the effort required to implement subprograms involves modifications of the AST classes.

2

Grammar Rules Relevant to Subprograms

- `subprogramDecls = ( subprogramDecl )* .`
- `subprogramDecl = procedureDecl | functionDecl .`
- `procedureDecl = "procedure" procId ( formalParameters )?`
  `"is" initialDeclPart statementPart procId ";" .`
- `functionDecl = "function" funcId ( formalParameters )?`
  `"return" typeName "is" initialDeclPart statementPart`
  `funcId ";" .`
- `formalParameters = "(" parameterDecl`
  `( "," parameterDecl )* ")" .`
- `parameterDecl = ( "var" )? paramId ":" typeName .`
- `procedureCallStmt = procId ( actualParameters )? ";" .`

3

Grammar Rules Relevant to Subprograms (continued)

- `actualParameters = "(" expressions ")" .`
- `returnStmt = "return" ( expression )? ";" .`
- `functionCall = funcId ( actualParameters )? .`

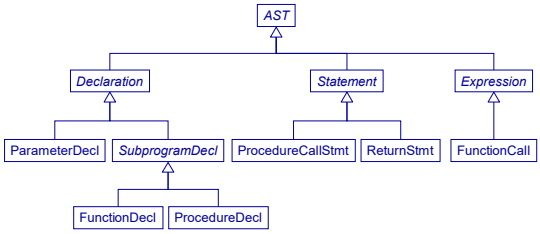
4

Relevant Parser Methods (based on grammar rules)

- `fun parseSubprogramDecls()    : List<SubprogramDecl>`
- `fun parseSubprogramDecl()     : SubprogramDecl?`
- `fun parseProcedureDecl()      : ProcedureDecl?`
- `fun parseFunctionDecl()       : FunctionDecl?`
- `fun parseFormalParameters()   : List<ParameterDecl>`
- `fun parseParameterDecl()      : ParameterDecl?`
- `fun parseProcedureCallStmt()  : ProcedureCallStmt?`
- `fun parseActualParameters()   : List<Expression>`
- `fun parseReturnStmt()         : ReturnStmt?`
- `fun parseFunctionCall()       : FunctionCall?`

5

Relevant AST Classes



6

Procedure Example – Parameters

```
var x : Integer;

procedure inc(var n : Integer) is
begin
   n := n + 1;
end inc;

begin
   x := 5;
   inc(x);
   writeln(x);
end.
```

What value is printed? If "var" is removed from the parameter declaration, what value is printed?

7

Procedure Example – Scope

```
var x : Integer;
var y : Integer;

procedure P1 is
   var x : Integer;
   var n : Integer;
begin
   x := 5;   // which x?
   n := y;   // which y?
end P1;

begin
   x := 8;      // which x?
end.
```

Variables and constants can be declared at the program (global) level or at the subprogram level, introducing the concept of scope.

8

The Scope Level of a Variable Declaration

- During code generation, when a variable or named value is referenced in the statement part of a program or subprogram, we need to be able to determine where the variable was declared.
- Class `IdTable` contains a method `getCurrentLevel()` that returns the block nesting level for the current scope.
  - `PROGRAM` for objects declared at the outermost (program) scope.
  - `SUBPROGRAM` for objects declared within a subprogram.
- When a variable is **declared**, the declaration is initialized with the current level.

```
val scopeLevel = idTable.getCurrentLevel()
val varDecl = VarDecl(identifiers, varType, scopeLevel)
```


9

Example: Scope Levels

```
var x : Integer;   // scope level of declaration is PROGRAM
var y : Integer;   // scope level of declaration is PROGRAM

procedure P1 is      // scope level or declaration is PROGRAM
   var x : Integer;  // scope level of declaration is SUBPROGRAM
   var b : Integer;  // scope level of declaration is SUBPROGRAM
begin
   ... x ...   // x was declared at SUBPROGRAM scope
   ... b ...   // b was declared at SUBPROGRAM scope
   ... y ...   // y was declared at PROGRAM scope
end p;

begin
    ... x  ...     //  x was declared at PROGRAM scope
    ... y  ...     //  y was declared at PROGRAM scope
    ... P1 ...     // P1 was declared at PROGRAM scope
end.
```


10

Class `IdTable`

- Class `IdTable` supports the ability to open new scopes and to search for declarations, both within the current scope and in enclosing scopes.
- Class `IdTable` is implemented as a stack of maps from identifier strings (names of things) to their declarations.
  (Note that, since we don't allow subprograms to be nested, our stack has at most two levels.)
- When a new scope is opened, a new map is pushed onto the stack. When a scope is closed, the top map is popped off the stack.

11

Class `IdTable` (continued)

- Within a subprogram, searching for a declaration involves searching within the current level (top map in the stack containing all identifiers declared at `SUBPROGRAM` scope) and then within the enclosing scope (the map under the top containing all identifiers declared at `PROGRAM` scope).

12

Selected Methods in the Modified Version of `IdTable`

```
/**
 * Opens a new scope for identifiers.
 */
fun openScope()

/**
 * Closes the outermost scope.
 */
fun closeScope()

/**
 * Add a declaration at the current scope level.
 * @throws ParserException if the identifier token associated
 *                         with the declaration is already
 *                         defined in the current scope.
 */
fun add(decl : Declaration)
```


13

Selected Methods in the Modified Version of `IdTable` (continued)

```
/**
 * Returns the Declaration associated with the identifier
 * token's text.  Returns null if the identifier is not found.
 * Searches enclosing scopes if necessary.
 */
operator fun get(idToken : Token) : Declaration?

/**
 * Returns the current scope level.
 */
fun getCurrentLevel() : ScopeLevel
```

Note: `ScopeLevel` is an `enum` class with only two values, `PROGRAM` and `SUBPROGRAM`.

14

Constraint Rules for Subprograms

- Return Statement
  - Type Rule: If the statement returns a value for a function, then the type of expression being returned must be the same as the function return type.
  - Miscellaneous Rule: If the return statement returns a value, then the return statement must be nested within a function declaration.
  - Miscellaneous Rule: If the return statement is nested within a function, then it must return a value.
  - Miscellaneous Rule: The return statement must be nested within a subprogram.*

*Handled by the parser using `SubprogramContext`.

15

Constraint Rules for Subprograms (continued)

- Function Declaration
  - Miscellaneous Rule: There should be no `var` parameters
  - Miscellaneous Rule: There should be at least one return statement.
  - Miscellaneous Rule: All return statements must return a value.
- Subprogram Call (for both procedures and functions)
  - Type Rule: The number of actual parameters should be the same as the number of formal parameters, and each corresponding pair of parameter types should match.
- Procedure Call
  - Miscellaneous Rule: If the formal parameter is a `var` parameter, then the actual parameter must be a named value (not an arbitrary expression).

16

Runtime Organization for Subprograms

Understanding the runtime organization for subprograms involves the following four major concepts

- Activation records
- Variable addressing
- Passing parameters and returning function values
- CVM instructions for subprograms

17

CVM Instructions for Subprograms

- `PROC` (procedure/function)
- `LDLADDR` (load local address)
- `LDGADDR` (load global address)
- `CALL` (call subprogram)
- `RET` (return from a subprogram)

Note that CVM does not have separate instructions for procedures and functions.

18

Active Subprograms

- When a program is running, a subprogram is said to be *active* if it has been called but has not yet returned.
- When a subprogram is called, we need to allocate space on the stack for its parameters and local variables. In addition, if the subprogram is a function, we need to allocate space on the stack for the return value.
- When the subprogram returns, the allocated stack space is released.

An active subprogram is one for which this space (activation record) is currently on the stack.

19

Activation Record
(a.k.a. Frame)

- An activation record is a run-time structure for each currently active subprogram. A new activation record is created every time a subprogram is called.
- Consists of up to five parts
  - return value part (for functions only)
  - parameter part (may be empty if there are no parameters)
  - context part (always 2 words)
    - saved values for PC and BP
  - local variable part (may be empty if there are no local variables)
  - temporary part
    - holds operands and results as statements are executed
    - is always empty at the beginning and end of every statement of the subprogram

20

Return Value Part of an Activation Record

- A function call must first allocate space on the stack for the return value. The number of bytes allocated is the number of bytes for the return type of the function.
- The `emit()` method in class `FunctionCall` contains the following code.

```
// allocate space on the stack for the return value
emit("ALLOC ${funcDecl.type.size}")
```


21

Parameter Part of an Activation Record

- For each *value* parameter, a subprogram call must emit code to leave the value of the actual parameter on the top of the stack.
- For each *variable* (`var`) parameter, a procedure call must emit code to leave the address of the actual parameter on the top of the stack. The actual parameter must be a named value, not an expression.

22

Context Part of an Activation Record

- Dynamic Link – base address (BP) of the activation record for the calling subprogram
- Return address – address of the next instruction following the call to the subprogram

The values for BP and PC relative to the calling subprogram are saved (pushed) onto the stack by the CVM "CALL" instruction, and they are restored by the CVM "RET" instruction.

23

Local Variable Part of an Activation Record

- If local variables are declared in a subprogram, then space must be allocated on the runtime stack for those variables.
- The CVM instruction `PROC` (procedure) has an integer argument for the variable length of subprogram.
- Example

```
procedure P2 is
   var m, n : Integer;
   var b : Boolean;
begin
   ...
end P2;
```


24

Local Variable Part of an Activation Record (continued)

- This procedure will need to allocate nine bytes for local variables. The instruction `PROC 9` will be emitted to allocate the necessary space on the runtime stack.
- An instruction of the form `PROC 0` is emitted whenever a subprogram does not have any local variables.

25

Temporary Part of an Activation Record

- The temporary part of an activation record is analogous to the use of the run-time stack to hold temporary values as described in Section 10.4.
- As machine instructions for a subprogram are executed, the temporary part grows and shrinks.
- The temporary part of an activation record is empty at the start and end of each CPRL statement in the subprogram.

26

Example: Temporary Part of an Activation Record

- Assume
  - register `BP` has the value `200`
  - local integer variable `x` has relative address `8`
  - local integer variable `y` has value `6` and relative address `12`
- The CPRL assignment statement

```
x := y + 1;
```

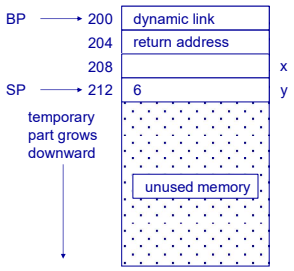
will compile to the following CVM instructions:

```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT 1 ←——— will be optimized to LDCINT1
ADD
STOREW
```


27

Example: Temporary Part of an Activation Record

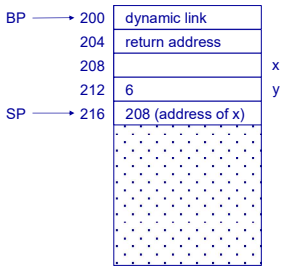
Temporary part is empty at the start of the CPRL statement.



28

Example: Temporary Part of an Activation Record (continued)

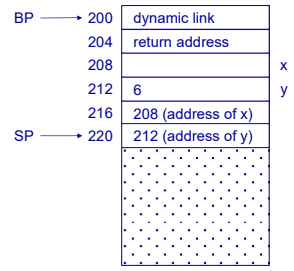
After execution of `LDLADDR 8`



29

Example: Temporary Part of an Activation Record (continued)

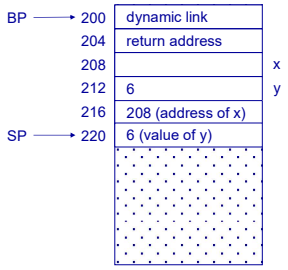
After execution of `LDLADDR 12`



30

Example: Temporary Part of an Activation Record (continued)

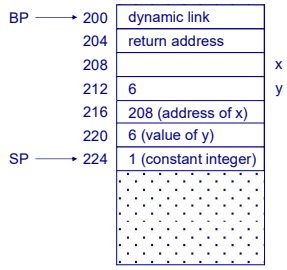
After execution of `LOADW`



31

Example: Temporary Part of an Activation Record (continued)

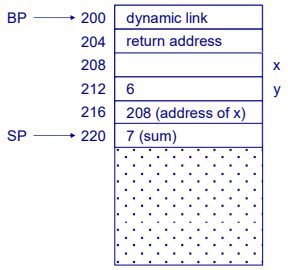
After execution of `LDCINT1`



32

Example: Temporary Part of an Activation Record (continued)

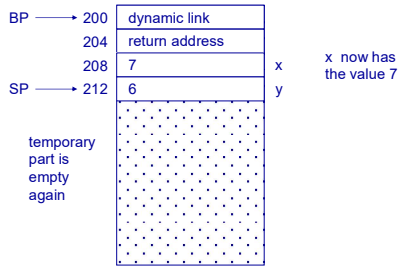
After execution of `ADD`



33

Example: Temporary Part of an Activation Record (continued)

After execution of `STOREW`



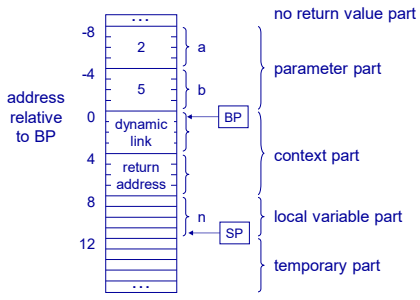
34

Example: Subprogram with Parameters

```
var x : Integer;

procedure P3(a : Integer, b : Integer) is
   var n : Integer;
begin
   ...
end P3;

begin
   ...
   P3(2, 5);
   ...
end.
```


35

Activation Record for Procedure `P3`



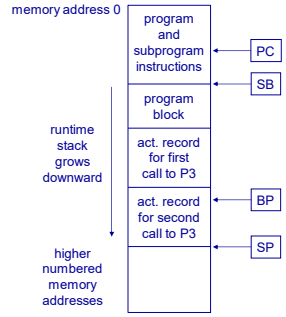
36

Supporting Recursion

- Since a new activation record is created every time a subprogram is called, CPRL supports recursive calls.
- To illustrate, suppose that a program calls procedure `P3`, and then `P3` makes a recursive call to itself. Each call to `P3` has its own activation record, which means each call has its own copy of parameters, locally declared variables, etc.
- The diagram on the next page illustrates this situation.

37

Supporting Recursion (continued)



38

Loading a Program

- The object code is loaded into the beginning of memory starting at address 0.
- Register `PC` is initialized to 0, the address of the first instruction.
- Register `SB` is initialized to the address following the last instruction (i.e., the first free byte in memory).
- Register `BP` is initialized to the address of the byte following the last instruction (i.e., the same as `SB`).
- Register `SP` is initialized to `BP` – 1 since the runtime stack is empty.

39

General Discussion of Parameters

- Functions can have only value parameters, but procedures can have both variable (var) and value parameters.
- The code to handle the passing of these two kinds of parameters as part of a procedure call is somewhat analogous to how you handle an assignment statement of the form "`x := y`", where you generate different code for the left and right sides.
  - Left side: generate code to leave the **address** on the stack.
  - Right side: generate code to leave the **value** on the stack.

40

General Discussion of Parameters (continued)

- Analogy for parameters
  - Variable (var) parameters: Generate code similar to the way you handle the left side of an assignment statement.
  - Value parameters: Generate code similar to the way you handle the right side of the assignment.
- When parsing the code for actual parameters, by default we always call `parseExpression()`.
  - generates code to leave the value of the expression on the stack
  - correct for a value parameter but not for a variable parameter
- Note that the code for class `Variable` contains a constructor that takes a single `NamedValue` object and uses it to construct a `Variable` object.

41

Converting `NamedValue` to `Variable`

- When you have a `NamedValue` expression corresponding to a variable parameter, you need to convert it to a `Variable`.
- One possible approach: In the `checkConstraints()` method of class `ProcedureCall`, when iterating through and comparing the list of formal parameters and actual parameters,
  - If the formal parameter is a variable parameter and the actual parameter is not a `NamedValue`, generate an error message (can't pass an arbitrary expression to a variable parameter).
  - If the formal parameter is a variable parameter and the actual parameter is a `NamedValue`, convert the `NamedValue` to a `Variable`.

42

Converting NamedValue to Variable (continued)

```
for (i in actualParams.indices)
  {
    var expr  = actualParams[i]
    val param = formalParams[i]

    ...  // check that types match

    ...  // check that named values are being passed
         // for var parameters (see next slide)
  }
```

(in method checkConstraints() of ProcedureCallStmt)

43

Converting NamedValue to Variable (continued)

```
// check that named values are being passed for var parameters
if (param.isVarParam)
  {
    if (expr is NamedValue)
      {
        // replace named value by a variable
        expr = Variable(expr)
        actualParams[i] = expr
      }
    else
      {
        val errMsg = "Expression for a var parameter must " +
                     "be a variable."
        throw error(expr.position, errMsg)
      }
  }
```

(in method checkConstraints() of ProcedureCallStmt)

44

Calling a Subprogram

When a subprogram is called

- For a function, space is allocated on the stack for the return value.
- The actual parameters are pushed onto the stack.
  - expression values for value parameters
  - addresses for variable parameters
- The CALL instruction pushes the context part onto the stack.
- The PROC instruction of the subprogram allocates space on the stack for the subprogram's local variables.

45

PROC Instruction versus ALLOC Instruction

- For CVM, the PROC instruction and the ALLOC instruction are equivalent and can be used interchangeably.
- Both instructions simply move SP to allocate space on the stack; e.g., for a function return value or a subprogram's local variable.

46

Return Instruction

- The CVM return instruction indicates the number of bytes used by the subprogram parameters so that they can be removed from the stack
- Example

```
ret 8
```


47

Returning from a Subprogram

When a return instruction is executed

- BP is set to the dynamic link.
  - restores BP to the caller's activation record
- PC is set to the return address.
  - restores PC to the caller's instructions
- SP is set so as to restore the stack to its state before the call instruction was executed.
  - For procedures, SP is set to the memory address before the activation record.
  - For functions, SP is set to the memory address of the last byte of the return value. The return value remains on the stack.

48

Referencing Local Variables and Parameters

The LDLADDR instruction is used to reference subprogram parameters and variables that are local to the subprogram.

- Computes the absolute address of a local variable from its relative address with respect to BP and pushes the absolute address onto the stack
- Use with subprograms is similar to the use of LDGADDR for program variables except that the relative address of the first local variable is 8 instead of 0 since there are 8 bytes in the context part of the activation record.
- Relative addresses can be negative to load the address of a parameter.

49

Referencing Global Variables

The LDGADDR instruction is used within a subprogram to reference global variables; i.e., variables declared at the program level.

- Computes the absolute address of a global variable from its relative address with respect to SB and pushes the absolute address onto the stack.
- Note that LDGADDR is used the same way in subprograms as it is in the main program.

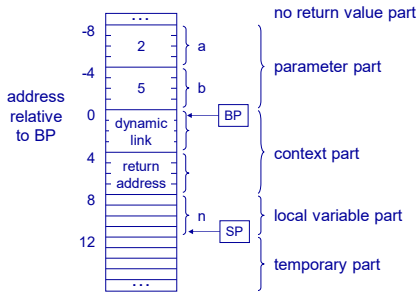
50

Example: Activation Record

```
var x : Integer;

procedure P3(a : Integer, b : Integer) is
   var n : Integer;
begin
   ...
end P3;

begin
   ...
   P3(2, 5);
   ...
end.
```


51

Activation Record for Procedure P3



52

Referencing Variables and Parameters for P3

- LDLADDR -8 loads (pushes) the address of parameter a onto the stack
- LDLADDR -4 loads (pushes) the address of parameter b onto the stack
- LDLADDR 8 loads (pushes) the address of local variable n onto the stack
- LDGADDR 0 loads (pushes) the address of global variable x onto the stack

53

Variable (var) Parameters

- For variable (var) parameters, the address of the actual parameter is passed; i.e., the value contained in the formal parameter is the address of the actual parameter.
- We need to use two instructions to load (push) the address of actual parameter onto the stack.

54

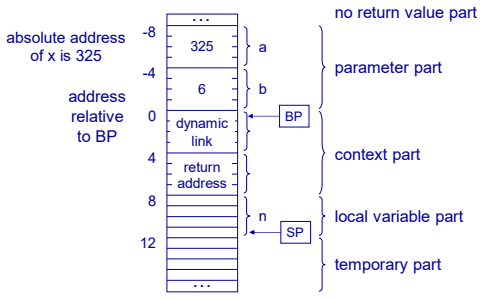
Example: Activation Record

```
var x : Integer;

procedure p4(var a : Integer, b : Integer) is
   var n : Integer;
begin
   ...
end p4;

begin
   x := 5;
   p(x, 6);
end.
```

Note that parameter a is a variable parameter.

55

Activation Record for Procedure P4
(after call p(x, 6))



56

Referencing Variables and Parameters for P4

- `LDLADDR -8` `LOADW` — loads (pushes) the address of the actual parameter x onto the run-time stack
- `LDLADDR -4` — loads (pushes) the address of parameter b onto the stack
- `LDLADDR 8` — loads (pushes) the address of local variable n onto the stack
- `LDGADDR 0` — loads (pushes) the address of global variable x onto the stack

57

Method emit() for Class Variable
(Loads the address of the variable onto the stack)

```
override fun emit()
  {
    if (decl is ParameterDecl && decl.isVarParam)
      {
        // address of actual parameter is value of var parameter
        emit("LDLADDR ${decl.relAddr}")
        emit("LOADW")
      }
    else if (decl.scopeLevel == ScopeLevel.PROGRAM)
        emit("LDGADDR ${decl.relAddr}")
    else
        emit("LDLADDR ${decl.relAddr}")
  }
```

We will extend this method again when we consider the topic of arrays in CPRL.

58