University of Toronto

CSC 488S / CSC2107S Compilers and Interpreters **Winter 2015/2016**

CSC488S Source Language Semantic Analysis

This handout describes the semantic analysis that should be performed on the project source language. The semantic analysis actions are described in terms of a set of semantic analysis actions **S**??

Semantic Analysis Rules

program:	**S**00 scope **S**01
statement:	variable ':' '=' expression **S**34 ,
	'**if**' expression **S**30 '**then**' statement ,
	'**if**' expression **S**30 '**then**' statement '**else**' statement ,
	'**while**' expression **S**30 '**do**' statement ,
	'**repeat**' statement '**until**' expression **S**30 ,
	'**exit**' **S**50 ,
	'**exit**' integer **S**50 **S**53 ,
	'**exit**' '**when**' expression **S**30 **S**50 ,
	'**exit**' integer '**when**' expression **S**30 **S**50 **S**53 ,
	'return' 'with' expression **S**51 **S**35 ,
	'**return**' **S**52 ,
	'write' output ,
	'read' input ,
	procedurename **S**42 ,
	procedurename '(' **S**44 arguments ')' **S**43 ,
	S06 scope **S**07 ,
	statement statement
declaration	'**var** variablenames ':' type **S**47 ,
	'function' functionname ':' type **S**11 **S**04 scope **S**05 **S**54 **S**13 ,
	'function' functionname **S**04 '(' **S**14 parameters ')' ':' type **S**12 scope **S**05 **S**54 **S**13 ,
	'procedure' procedurename **S**17 **S**08 scope **S**09 **S**13 ,
	'procedure' procedurename **S**08 '(' **S**14 parameters ')' **S**18 scope **S**09 **S**13 ,
	declaration declaration
variablenames:	variablename **S**10 ,
	variablename '[' bound ']' **S**19 ,
	variablename '[' bound ',' bound ']' **S**19 ,
	variablenames ',' variablenames
bound	integer ,
	generalBound '.' '.' generalBound **S**46
generalBound	integer ,
	'-' integer
scope	'{' declaration **S**02 statement '}' ,
	'{' statement '}' ,
	'{' '}'

output:	expression **S**31 ,
	text ,
	’newline’ ,
	output ’,’ output
input:	variable **S**31 ,
	input ’,’ input
type:	**’integer’** **S**21 ,
	’boolean’ **S**20
arguments:	expression **S**45 **S**36 ,
	arguments ’,’ arguments
parameters:	parametername ’:’ type **S**16 **S**15 ,
	parameters ’,’ parameters
variable:	variablename **S**26 ,
	arrayname ’[’ expression **S**31 ’]’ **S**27
	arrayname ’[’ expression **S**31 ’,’ expression **S**31 ’]’ **S**27
expression:	integer **S**21 ,
	’-’ expression **S**31 **S**21 ,
	expression **S**31 ’+’ expression **S**31 **S**21 ,
	expression **S**31 ’-’ expression **S**31 **S**21 ,
	expression **S**31 ’*’ expression **S**31 **S**21 ,
	expression **S**31 ’/’ expression **S**31 **S**21 ,
	’true’ **S**20 ,
	’false’ **S**20 ,
	’not’ expression **S**30 **S**20
	expression **S**30 **’and’** expression **S**30 **S**20 ,
	expression **S**30 **’or’** expression **S**30 **S**20 ,
	expression ’=’ expression **S**32 **S**20 ,
	expression **’not’** ’=’ expression **S**32 **S**20 ,
	expression **S**31 ’<’ expression **S**31 **S**20 ,
	expression **S**31 ’<’ ’=’ expression **S**31 **S**20 ,
	expression **S**31 ’>’ expression **S**31 **S**20 ,
	expression **S**31 ’>’ ’=’ expression **S**31 **S**20 ,
	’(’ expression ’)’ **S**23 ,
	’(’ expression **S**30 ’?’ expression ’:’ expression **S**33 ’)’ **S**24 ,
	variable ,
	functionname **S**42 **S**28 ,
	functionname ’(’ **S**44 arguments ’)’ **S**43 **S**28 ,
	parametername **S**25 ,
variablename:	identifier **S**37
arrayname:	identifier **S**38
functionname:	identifier **S**40
procedurename:	identifier **S**41
parametername:	identifier **S**39

Semantic Analysis Operators

Scopes and Program

These semantic operators are used to keep track of scopes in the program being compiled.

S00	Start program scope.
S01	End program scope.
S02	Associate declaration(s) with scope.
S04	Start function scope.
S05	End function scope.
S06	Start ordinary scope.
S07	End ordinary scope.
S08	Start procedure scope.
S09	End procedure scope.

Declarations

These semantic operators make entries in the symbol table for the current scope. All of the *Declare...* operators should check that the identifier being declared has not already been declared in the current scope.

S10	Declare scalar variable.
S11	Declare function with no parameters and specified type.
S12	Declare function with parameters and specified type.
S13	Associate scope with function/procedure.
S14	Set parameter count to zero.
S15	Declare parameter with specified type.
S16	Increment parameter count by one.
S17	Declare procedure with no parameters.
S18	Declare procedure with parameters.
S19	Declare array variable with specified lower and upper bounds.
S46	Check that lower bound is <= upper bound.
S47	Associate type with variables.

Statement Checking

These semantic operators check various correctness conditions for statements.

S50	Check that **exit** statement is directly inside a loop.
S51	Check that **return** is directly inside a function
S52	Check that **return** statement is directly inside a procedure.
S53	Check that integer is > 0 and <= number of containing loops.
S54	Check that function body contains at least one **return** statement

Expressions Types

These semantic operators are used to keep track of the type of expressions. In the model used in this hand-out, a type (integer or boolean) is a associated with the left hand side of each rule in the expression part of the grammar. The *Set result type to ...* semantic operators (somehow) associate a type with the left hand side. This same mechanism is used to keep track of types in declarations.

S20 Set result type to boolean.
S21 Set result type to integer.
S23 Set result type to type of expression.
S24 Set result type to type of conditional expressions.
S25 Set result type to type of parametername.
S26 Set result type to type of variablename.
S27 Set result type to type of array element.
S28 Set result type to result type of function.

Expression Type Checking

These semantic operators check that the type of an expression is correct for the use that is being made of the expression.

S30 Check that type of expression is boolean.
S31 Check that type of expression or variable is integer.
S32 Check that left and right operand expressions are the same type.
S33 Check that both result expressions in conditional are the same type.
S34 Check that variable and expression in assignment are the same type, and that the assignment is valid.
S35 Check that expression type matches the return type of enclosing function.
S36 Check that type of argument expression matches type of corresponding formal parameter.
S37 Check that identifier has been declared as a scalar variable.
S38 Check that identifier has been declared as an array.
S39 Check that identifier has been declared as a parameter.

Functions, procedures and arguments

These semantic operators are used the check that procedures and functions are being used correctly.

S40 Check that identifier has been declared as a function.
S41 Check that identifier has been declared as a procedure.
S42 Check that the function or procedure has no parameters.
S43 Check that the number of arguments is equal to the number of formal parameters.
S44 Set the argument count to zero.
S45 Increment the argument count by one.

Complete Semantic Processing

Your semantic analyzer should attempt to find all of the semantic errors in the program it is processing. Stopping after the first error is not acceptable.

Addressing for Variables

The pseudo machine that you will be generating code for uses a simple form of addressing for variables and parameters called *lexic level , displacement* addressing, (similar to base register + displacement addressing found on many modern machines).

An address is a pair of numbers: (`lexic level` , `displacement`) where:

lexic level is the *static* depth of nesting of scopes in the program. The main program is lexic level zero, scopes directly inside the main program are lexic level one, etc. It is your design choice whether the lexic level gets incremented for all scopes or only for *major scopes* (program, functions and procedures), although only incrementing lexic level for major scopes is strongly recommended.

displacement The storage for variables in a scope is laid out as a block of consecutive memory locations. The algorithm to do this is one of your design choices. For a variable, the *displacement* of the variable is its offset relative to the start of the block of storage for its scope. The pseudo machine uses *word addressing* so consecutive memory locations are addressed by consecutive integer displacements.

It is really convenient to setup this basis for addressing variables during semantic analysis when the scopes in a program are being identified and the declarations in each scope are being processed. Every variable and parameter needs to be associated with a scope (its *lexic level*) and its location in the scope (its *displacement*.

Functions and procedures are addressed using the memory address of their first instruction. Addressing for functions and procedures will be discussed in the forthcoming code generation handout.

The meaning of ”directly inside”

The definition of the semantic checks **S**50 , **S**51 and **S**52 uses the phrase *directly inside*. The intent is to disallow misuse of nested constructs that would otherwise be legal. Checking for these restrictions should restart at procedure/function boundaries. Examples:

```
while K <= 100 do {
    procedure P {
        % violates S50
        exit when K = 47
    } % end of P
    ...
} % end of loop
```

```
function F : integer {
    procedure Q {
        % violates S51
        return with 42
    } % end of Q
    ...
} % end of F
```

```
procedure R { % all legal
    while K not = 42 {
        if K = 23 then
            { ... exit }
        else
            { ... return }
    }       } % end R
```

No Change to Project Language Syntax

Nothing in this document is intended to change the syntax of the course project language. If you discover a case where the syntax of the language in this document differs from the Source Language Reference Grammar, it is an error in this document, not an intentional change. Please notify the instructor if you think there is an error in this document.