University of Toronto

CSC 488S Compilers & Interpreters **Winter 2013/2014**

CSC488S Source Language Semantic Analysis

This document 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 '**fi**' ,
	'**if**' expression **S**30 '**then**' statement '**else**' statement '**fi**' ,
	'**while**' expression **S**30 '**do**' statement '**end**' ,
	'**repeat**' statement '**until**' expression **S**30 ,
	'**exit**' **S**50 ,
	'**exit**' '**when**' expression **S**30 **S**50 ,
	'**result**' expression **S**51 **S**35 ,
	'**return**' **S**52 ,
	'**put**' output ,
	'**get**' input ,
	procedurename '(' **S**44 argumentList ')' **S**43 ,
	S06 scope **S**07 ,
	statement statement
scope	'{' declaration **S**02 statement '}' ,
	'{' statement '}'
	'{' '}'
declaration:	'**var**' variablenames ':' type **S**47 ,
	functionHead **S**49 **S**04 **S**54 scope **S**05 **S**13 ,
	procedureHead **S**49 **S**08 **S**54 scope **S**09 **S**13 ,
	'**forward**' functionHead **S**11
	'**forward**' procedureHead **S**17
	declaration declaration
functionHead:	'**func**' functionname '(' **S**14 parameterList ')' ':' type **S**12
procedureHead:	'**proc**' procedurename '(' **S**14 parameterList ')' **S**18
variablenames:	variablename **S**10 ,
	variablename '[' bound ']' **S**19 ,
	variablename '[' bound ',' bound **S**46 ']' **S**48 ,
	variablenames ',' variablenames
bound	integer ,
	generalBound '.' '.' generalBound
generalBound	integer ,
	'-' integer ,

type:	'**integer**' **S**21 ,
	'**boolean**' **S**20
output:	expression **S**31 ,
	text ,
	'**newline**' ,
	output ',' output
input:	variable **S**31 ,
	input ',' input
argumentList:	arguments ,
	% EMPTY
arguments:	expression **S**45 **S**36 ,
	arguments ',' arguments
parameterList:	parameters ,
	% EMPTY
parameters:	parametername ':' type **S**16 **S**15 ,
	parameters ',' parameters
variable:	variablename **S**26 ,
	parametername **S**25 ,
	arrayname '[' expression **S**31 ']' **S**38 **S**29 **S**27
	arrayname '[' expression **S**31 ',' expression **S**31 ']' **S**55 **S**29 **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**44 argumentList ')' **S**43 **S**28 ,
variablename:	identifier **S**37 **S**29
arrayname:	identifier
functionname:	identifier **S**40 **S**29
procedurename:	identifier **S**41 **S**29
parametername:	identifier **S**39 **S**29

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 statement scope.
S07	End statement 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 forward function .
S12	Declare function with parameters (if any) 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 forward procedure.
S18	Declare procedure with parameters (if any).
S19	Declare one dimensional array with specified bound.
S46	Check that lower bound is <= upper bound.
S47	Associate type with variables.
S48	Declare two dimensional array with specified bounds.
S49	If function/procedure was declared forward, verify forward declaration matches.
S54	Associate parameters if any with scope.

Statement Checking

These semantic operators check various correctness conditions for statements.

S50	Check that **exit** statement is inside a loop.
S51	Check that **result** statement is directly inside a function
S52	Check that **return** statement is directly inside a procedure.

Expressions Types

These semantic operators are used to keep track of the type of expressions. In the model used in this document, 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 expressions in conditional are the same type.
S34	Check that variable and expression in assignment are the same type.
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 arrayname has been declared as a one dimensional array.
S39	Check that identifier has been declared as a parameter.
S55	Check that arrayname has been declared as a two dimensional array
S29	Check that identifier is visible according to the language scope rule.

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.
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

The semantic analyzer that you build should find all of the semantic errors in the program it is processing. Stopping after finding 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 , order number* addressing.

An address is a pair of numbers: `( lexic level , order number )` 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). Only for major scopes is strongly recommended.

order number is an integer that uniquely identifies each variable declared in a scope. Conventionally the first variable declared in a scope is assigned order number zero, the second order number one, etc. This is particularly convenient for addressing on the pseudo machine that you will be generating code for.

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 *order number*.

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 document.

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.