COMPILER DESIGN LABORATORY

SOURCE LANGUAGE DESCRIPTION

1. Simple Integer Language (SIL)

The language specification given here is informal and gives a lot of flexibility for the designer to write the grammatical specifications to his/her own taste. The following features are the minimal requirements for the language.

2. General Program Structure

Global Declarations

Function Definitions

Main Function Definition

3. Global Declarations

The global declaration part of a SIL program begins with the keyword **decl** and ends with the keyword **enddecl**. Declarations should be made for global variables and functions defined in the SIL program.

Global variables may be of type *Integer, Boolean, Integer array* or *Boolean array*. The variables declared globally must be allocated statically. Boolean variables can hold only the special Boolean constants –**TRUE/FALSE**. Global variables are visible throughout the program **unless suppressed** by a redeclaration within the scope of some function. Array type variables can be declared only globally. Only single dimensional arrays are allowed. Variables **cannot** be assigned values during the declaration phase.

For every function except the ***main*** function defined in a SIL program, there must be a declaration. A function declaration should specify the name of the function, the name and type of each of its arguments and the return type of the function. A function can have integer/boolean arguments. Parameters may be passed by **value** or **reference**. Arrays **cannot** be passed as arguments. If a global variable name appears as an argument, then within the scope of the function, the new declaration will be valid and global variable declaration is **suppressed**. Different functions may have arguments of the same name. However, the same name cannot be given to different arguments in a function. The return type of a function must be either integer or Boolean. The general form of declarations is as follows:

Type VarName/FunctionName [ArraySize]/(ParameterList); //Third part needed only for arrays/functions

Example:

decl

integer x,y,a[10],b[20]; // x,y are integers, a,b are integer arrays

integer f1(integer a1,a2; boolean b1; integer &c1), f2(); // c1 is passed by *reference*, rest by *value*

boolean t, q[10], f3(integer x); // variable, array and a functions declared together

integer swap(integer &x, &y); // x, y are passed by reference

enddecl // Please note the use of "**,**" and "**;**"

Declaring functions at the beginning avoids the "forward reference" problem and facilitates simpler single pass compilation. Note that the declaration syntax of functions is structurally same as that for variables. Finally, inside **swap**, the global variables **x** and **y** are no more visible because of the re-declaration and global declaration for **x** is suppressed in **f3**. If a variable/function is declared twice at the same point, a compilation error should result.

4. Function Structure and Local Variables

All globally declared variables are visible inside a function, unless suppressed by a re-declaration. Variables declared inside a function are **invisible** outside. The general form of a function definition is given below:

<Type> FunctionName(ArgumentList) {

Local Declarations

Function Body

}

The arguments and return type of each function definition should match exactly with the corresponding declaration. Argument names must be type checked for **name equivalence** against the declaration.*Every declared function must have a definition*. The compiler should report error otherwise.

The syntax of local declarations and definitions are similar to those of global declarations ***except that arrays and functions cannot be declared inside a function***.

Local variables are visible only within the scope of the function where they are declared. Scope rules for parameters are identical to those for variables.

The **main()** function, by specification, must be a **zero argument** function of type **integer**. *Program execution begins from the body of the main function.* The main function *need not* be declared. The definition part of main should be given in the same format as any other function.

The **Body** of a function is a collection of statements embedded within the keywords **begin** and **end.**

Example: The definition of **swap** declared above may look like the following:

integer swap **(integer** &x, &y**) {**

decl

integer q **//** q is re-declared causing suppression of global declaration

enddecl

begin

q = x;

x = y;

y = q; // Note the syntax for using variables passed by reference.

return 1; // swap must return an integer.

end // Note that SIL doesn't support void functions.

}

Local Variables and parameters should be allocated space in the **run-time stack**. The language supports recursion and **static scope rules apply**.

5. Main and Function Body

A Body is a collection of **statements** embedded within the keywords **begin** and **end.** Each statement should end with a ';' which is called the **terminator**. There are five types of statements in SIL. They are:

a.Assignment Statement b) Conditional Statement

a.Iterative statement d) Return statement e) Input/Output

Before taking up statements, we should look at the different kinds of **expressions** supported by SIL.

6. Expressions

SIL has two kinds of expressions, a) Arithmetic and b) Logical

6.1 Arithmetic Expressions

Any constant, integer variable or an indexed array variable is a SIL expression provided the expression is within the scope of the concerned variable declarations. SIL treats a function as an expression and the value of a function is its return value. SIL supports **recursion**.

SIL provides five arithmetic operators, viz., +, -, *, / (Integer Division) and % (Modulo operator) through which arithmetic expressions may be combined. Expression syntax and semantics are similar to standard practice in programming languages and normal rules of precedence, associativity and paranthesization hold. SIL is strongly typed and ***any type mismatch must be reported at compile time***.

Examples: 5, a[a[5+x]]+x , (f2() + b[x] + 5) etc. are arithmetic expressions.

6.2 Logical Expressions

Logical expressions can take values TRUE or FALSE. Logical expressions may be formed by combining arithmetic expressions using **relational operators**. The relational operators supported by SIL are <, >, <=, >=, ==, and !=. Again standard syntax and semantics conventions apply. TRUE and FALSE are constant logical expressions. Every boolean variable is a logical expression and its value is the value stored in its location. Logical expressions themselves may be combined using **logical operators** AND, OR and NOT. Logical expressions may be assigned to boolean variables only. Note that a relational operator can compare only two arithmetic expressions and not two logical expressions. Similarly, a logical operator can connect only two logical expressions (except for NOT which is a unary logical operator).

Example:((x==y)==a[3]) is **not valid** SIL expression because (x==y) is a logical expression, while a[3] is an arithmetic expression and "==" operates only between two arithmetic expressions.

7. Assignment Statement

The SIL assignment statement assigns the value of an expression to a variable, or an indexed array of the **same type**. Type errors must be reported at compile time. The general syntax is as the following:

<**Variable**> = <**Expression**>;

Example: q[3]=(x==y); t=TRUE; are both valid assignments to boolean variables.

8. Conditional Statement

The SIL conditional statement has the following syntax:

if <Logical Expression> **then**

Statements

else

Statements

endif;

The **else** part is optional. The statements inside an **if**-block may be conditional, iterative, assignment, or input/output statements, but **not** the return statement.

9. Iterative Statement

The SIL iterative statement has the following syntax:

while < Logical Expression > **do**

Statements

endwhile;

Standard conventions apply in this case too. The statements inside a **while**-block may be conditional, iterative, assignment, or input/output statements, but **not** the return statement.

10. Return Statement

The main body as well as each function body should have **exactly one** return statement and it should be the **last statement** in the body. The syntax is:

return <Expression> **;**

The return value of the function is the value of the expression. The return type should match the type of the expression. Otherwise, a compilation error should occur.

The **return type of main is integer** by specification.

11. Input/Output

SIL has two I/O statements – **read** and **write**. The syntax is as the following:

read (<IntegerVariable>**);**

write (<Arithmetic Expression>**);**

The *read* statement reads an integer value from the standard input device into an

integer variable or an indexed array variable. The *write* statement outputs the value of the arithmetic expression into the standard output. Note that *input/output operations are not allowed on boolean type*.

Example: **read (a [x]);**

write (7*(5+a[9]);

12. An Example SIL Program

The following SIL program calculates and prints out the factorial of the first ***n*** numbers, value of ***n*** read from standard input.

```
decl
        integer factorial(integer n);
enddecl
integer factorial (integer n) {
decl
        integer rvalue;
enddecl
begin
        if (n==1) then
                rvalue = 1;
        else
                rvalue = n * factorial (n-1);
        endif;
        return rvalue; // Note only one RETURN statement is allowed.
end
}
integer main( ){ // Main definition should always begin like this
decl
        integer n,i ;
enddecl
```

```
begin
        read (n);
        i = 1;
        while ( i <= n) do
                write ( factorial(i));
                i = i + 1;
        endwhile;
        return 1; // Any integer value may be returned
end
}
```

ESIL: Extended SIL

There are three important extensions to the language features that must be considered. Some implementation issues are also discussed. This part of the documentation is better read after implementing SIL compiler.

a) **Introducing nested functions:** This means provision to declare and define functions inside functions. The scope rules for functions are similar to those for variables. **Function parameters** also must be supported (This makes the language similar to PASCAL in block structure). Nested functions would require the use of **static links** to be stored in the activation record of a function. Moreover, the *symbol table structure must carry the nesting information*. The simplest implementation would be to have one symbol table for each layer of nesting during compilation of a function. Unlike SIL, where there are only two layers of nesting (global and local), **the number of layers of nesting in a ESIL program can be determined only during compilation.** Note that **static scope** rules apply.

b) **Providing User defined types**: User defined types may be supported by allowing creation of compound data types from simple or already defined data types using the **typedef** statement. All type definitions should be given **before** global declarations. The syntax is as follows:

```
typedef name1{
        integer x;
        boolean y;
```

}

The member **fields** of a newly defined type may be of type **integer**, **boolean** or a **previously defined type**. Arrays are **not** allowed

typedef *name2* {

name1 g;

boolean t;

}

Once a type is defined, variables of the type may be defined in the usual manner. The type declaration is visible throughout the program.

decl

name2 w[10], u;

integer temp;

enddecl

The following statement illustrates the access to a variable of a user-defined type.

temp = w[5]**.**g**.**x; // The dot operator is used to address the fields.

W[5] = u; // ***Name equivalence*** of types must be checked here.

For each user-defined type, a **type expression tree** must be created at the time of parsing the type definition. The relative addresses or **offsets** of each field element (from the starting storage location of a variable of that type) can be fixed at the time of type definition and may be stored in the expression tree itself. The symbol table entry for a variable shall contain a pointer to the corresponding type expression tree.

c) **Supporting Dynamic Memory:** This requires adding pointer data type to the language. A variable may defined as a pointer to a primitive/user-defined type as follows:

decl

name2 ^p1; // p1 is a pointer to type name

integer ^p2; // p2 is a pointer to type integer

enddecl

Dynamic allocation and de-allocation may be achieved through implementing the library functions **new** and **free**. The calling syntax illustrated by the following

example:

pointer_var = new(*size*); // size must be **integer**

free(pointer_var)

Let the number of storage locations required for storing a variable of the type pointed to of Pointer_var be k. Then, *size x k* **contiguous** locations in memory will be allocated and Pointer_var shall contain the starting address of the memory allocated. Once allocated, the pointer variable can be used like an array. The use of pointer variables is illustrated by the following example:

p1= new(10); // Now p1 is allocated and can be used like an array of 10 elements

p1[4]**.**g**.**x = 5;

free(p1) // Free the previous allocation.

new and **free** may be implemented by declaring a large array and using it as a heap. (Refer to any standard text on Data Structures for heap management algorithms.) The tricky part is that ***new*** *and* ***free*** *will have to be written in SIL!* (Alternate way is to write them in SIM assembly language). A neat way of doing it would be to assemble them as a separate library. The compiler can be adapted to append the library code to the target code it generates.

Please send Error Reports/Comments/Queries about this document
to kmurali@nitc.ac.in