University of Arizona, Department of Computer Science

CSc 453 — Assignment 3 — Due Wed Oct 28, 23.59 — 10%

Christian Collberg
October 11, 2009

1 Introduction

Your task is to write a semantic analyzer for the language Luca. Your program should be named `luca_sem`. `luca_sem` reads a source program, does lexical, syntactic, and semantic analysis and writes semantic error messages to `standard error`. A semantically correct program will produce no output.

- Your tree-walk evaluator should be programmed in an *applicative* style. I.e., all information should be passed around the tree using synthesized, inherited, and threaded attributes; there should be no global variables. In particular, you should use symbol tables and environments to represent scope information, as shown in lectures. The tree-walk evaluator should make exclusive use of recursion; iteration is not allowed. If you do make use of iteration and global data, points will be deducted.
- This assignment can be coded in the language of your choice as long as it can be compiled and run on lectura.
- Make sure that your `Makefile` is working properly, and that `luca_sem` is called exactly as in the example above.
- You should not exit the semantic analyzer after the first error. Rather, you should generate as complete error messages as possible.
- You should avoid cascading error messages. Consider, for example, the expression $x+(y+3.14)$ where x and y are both declared as integers. The subexpression $(y+3.14)$ should generate a type error since integers can't be added to reals. However, $x+(\ldots)$ should not produce any "extra" messages as a result of the subexpression being erroneous.
- The error messages should be of the form

  ```
  <SEMANTIC_ERROR pos="3" message="Identifier expected" argument="X"/>
  ```

  and should be written to standard output.
- The syntax is the same as in the previous assignment.

2 Semantic rules

- Identifiers have to be declared before they are used.
- Identifiers cannot be redeclared in the same scope.
- There are four (incompatible) built-in types, `INTEGER`, `REAL`, `BOOLEAN` and `CHAR`.

- The identifiers `TRUE` and `FALSE` are predeclared in the language.
- Identifiers are case sensitive.
- Here are the type rules for Luca expressions:

Left	Operators	Right		Result
Int	‘+’, ‘−’, ‘∗’, ‘/’, ‘%’	Int	⇒	Int
Real	‘+’, ‘−’, ‘∗’, ‘/’	Real	⇒	Real
Int	‘<’,‘<=’, ‘=’, ‘#’, ‘>=’, ‘>’	Int	⇒	Bool
Real	‘<’,‘<=’, ‘=’, ‘#’, ‘>=’, ‘>’	Real	⇒	Bool
Char	‘<’,‘<=’, ‘=’, ‘#’, ‘>=’, ‘>’	Char	⇒	Bool
Bool	‘**AND**’, ‘**OR**’	Bool	⇒	Bool
	‘**NOT**’	Bool	⇒	Bool
	‘−’	Int	⇒	Int
	‘−’	Real	⇒	Real
	‘**TRUNC**’	Real	⇒	Int
	‘**FLOAT**’	Int	⇒	Real

- As seen from the table above, Luca does not allow *mixed arithmetic*, i.e. there is no *implicit conversion* of integers to reals in an expression. For example, if `I` is an integer and `R` is real, then `R:=I+R` is illegal.
- Luca instead supports two explicit conversion operators, `TRUNC` and `FLOAT`. `TRUNC R` returns the integer part of `R`, and `FLOAT I` returns a real number representation of `I`.
- Note that `%` (remainder) is not defined on real numbers.
- For a constant expression, division by 0 isn't allowed.
- Arithmetic operations on characters are not allowed.
- Variables are not allowed in constant expressions.
- Only reals, integers, and characters can be read or written.
- The left hand side of an assignment statement and the designator in a `READ` statement must be writeable (i.e. an L-value).
- EXIT statements can only occur within LOOP statements.
- A procedure's formal parameters and local declarations form one scope, which means that it is illegal for a procedure to have a formal parameter and a local variable of the same name. Parameters are passed by value unless the formal parameter has been declared **VAR**. Only L-valued expressions (such as ‘`A`’ and ‘`A[5]`’) can be passed to a **VAR** formal.
- The assignment `A:=B` is illegal if `A` or `B` are records or arrays.
- Procedures can be nested.

3 Context Conditions

Below are the error conditions you need to check, organized by AST node. In some cases the order in which you check the conditions matters, particularly if you want to generate the same messages as I do! The reason is that we want to avoid cascading error messages — the test you perform first is more likely to produce an error message than one you perform later.

- At the end of semantic analysis I *sort* all my error messages lexicographically. If you do the same we're more likely to generate the same sequences of errors.

DECL:

An identifier can only be declared once in each scope. A procedure's formal parameters and local declarations form one scope. If `ID` is declared more than once, issue this error message:

```
<SEMANTIC_ERROR pos="..." message="Multiple declaration" argument="ID"/>
```

VARDECL,FIELDDECL,FORMALDECL:

1. The type name must be declared:
   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```
2. And, if the type name is declared, it has to be declared to be a type:
   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```

CONSTDECL:

1. The type name must be declared:
   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```
2. And, if the type name is declared, it has to be declared to be a type:
   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```
3. And, if it's declared a type, it has to be declared a *scalar* type (integer, character, real, boolean):
   ```
   <SEMANTIC_ERROR pos="..." message="Scalar type expected"/>
   ```
4. If the declared type is OK, you need to check that the expression is of the same type:
   ```
   <SEMANTIC_ERROR pos="..." message="Wrong expression type"/>
   ```
5. Regardless of the type checks above, the expression has to be constant-valued:
   ```
   <SEMANTIC_ERROR pos="..." message="Constant expression expected"/>
   ```

ARRAYDECL:

1. The type name must be declared:
   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```
2. And, if the type name is declared, it has to be declared to be a type:
   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```
3. The array size must be of type integer:
   ```
   <SEMANTIC_ERROR pos="..." message="Integer expression expected"/>
   ```
4. Regardless, of its type, the array size must be a constant expression:
   ```
   <SEMANTIC_ERROR pos="..." message="Constant expression expected"/>
   ```

ASSIGN:

1. The left hand and the right hand side must be of scalar (integer, real, char, boolean) type.
   ```
   <SEMANTIC_ERROR pos="..." message="Scalar type expected"/>
   ```
2. The left hand and the right hand side must be the same type.
   ```
   <SEMANTIC_ERROR pos="..." message="Type missmatch in assignment statement"/>
   ```
3. The left hand side must be a L-value, i.e. something you can assign to.
   ```
   <SEMANTIC_ERROR pos="..." message="Can't assign to a constant"/>
   ```

PROCCALL:

1. The designator must be a single declared identifier:
   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared"/>
   ```
2. If the identifier is declared, it must be declared to be a procedure:
   ```
   <SEMANTIC_ERROR pos="..." message="Procedure identifier expected"/>
   ```

WRITE:
The expression must evaluate to an integer, real, character, or string.

```
<SEMANTIC_ERROR pos="..." message="INTEGER, REAL, CHAR, STRING type expected"/>
```

READ:

1. The designator must evaluate to an integer, real, or character:
   ```
   <SEMANTIC_ERROR pos="5" message="INTEGER, REAL, CHAR type expected"/>
   ```
2. The designator has to be an L-value (i.e. something you can assign to):
   ```
   <SEMANTIC_ERROR pos="..." message="Can't read to a constant"/>
   ```

WHILE,REPEAT,IF1,IF2:
The expression must be a boolean:

```
<SEMANTIC_ERROR pos="..." message="Boolean type expected"/>
```

EXIT:
`EXIT` must not occur outside of a loop:

```
<SEMANTIC_ERROR pos="..." message="EXIT only within LOOP"/>
```

ACTUAL:

1. There have to be the same number of actual and formal parameters:
   ```
   <SEMANTIC_ERROR pos="..." message="Too many actual parameters"/>
   <SEMANTIC_ERROR pos="..." message="Too few actual parameters"/>
   ```
2. Regardless, the actual parameter has to be assignable to the corresponding formal parameter.

```
<SEMANTIC_ERROR pos="..." message="Actual/formal parameter type missmatch"/>
```

3. Regardless, if a formal parameter is declared to be a **VAR** parameter, then the corresponding actual has to be an L-value (cannot be a constant):

   ```
   <SEMANTIC_ERROR pos="..." message="VAR formal parameter requires variable actual"/>
   ```

VARREF:

1. The identifier has to be declared:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="ID"/>
   ```

2. The identifier must be a formal parameter, a variable (global or local), a constant identifier, or a procedure:

   ```
   <SEMANTIC_ERROR pos="..." message="Variable expected"/>
   ```

INDEX:

1. The index expression must evaluate to an integer type:

   ```
   <SEMANTIC_ERROR pos="..." message="Integer type expected"/>
   ```

2. The designator must be of array type:

   ```
   <SEMANTIC_ERROR pos="..." message="Array variable expected"/>
   ```

FIELDREF:

1. The designator must be of record type:

   ```
   <SEMANTIC_ERROR pos="..." message="Record variable expected"/>
   ```

2. If the designator is or record type, then the field must be declared in the record:

   ```
   <SEMANTIC_ERROR pos="..." message="Field identifier not declared" argument="ID"/>
   ```

BINARY,UNARY:
The table in the previous section gives the semantic rules for expressions. For constant expressions, division by zero isn't allowed.

1. In arithmetic expressions, when a real or integer is expected, issue:

   ```
   <SEMANTIC_ERROR pos="..." message="Numeric type expected"/>
   ```

2. For $a\%b$, if a and b aren't integer types, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Integer type expected"/>
   ```

3. For `AND`, `OR`, `NOT`, if the arguments aren't boolean types, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Boolean type expected"/>
   ```

4. When the arguments to comparison operators (`#, <, >, <=, >=, =`) aren't integers, reals, booleans, or chars, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Scalar or reference type expected"/>
   ```

   (Reference type refers to another version of Luca that also has pointer types.)

5. For `TRUNC` and `FLOAT`, respectively, when the arguments are of the wrong type, issue

```
<SEMANTIC_ERROR pos="..." message="Real type expected"/>
<SEMANTIC_ERROR pos="..." message="Integer type expected"/>
```

6. Otherwise, if the left and right hand sides are of different types, issue:

```
<SEMANTIC_ERROR pos="..." message="Type missmatch"/>
```

4 Getting Started

Here's what I would do to get started with the assignment:

1. Write a `Symbol` symbol class. This might look something like this:

```
class Symbol {
   String name;
   int pos;
   int number;
}

class Variable extends Symbol {
   Symbol type;
}

class Constant extends Symbol {
   Symbol type;
   Value value;
}

class Type extends Symbol { ... }

class BasicType extends Type { ... }

class ArrayType extends Type { ... }

class Procedure extends Symbol { ... }
```

   I.e. for every type of symbol you need to be able to store all necessary data about that symbol.

2. Write a `SyTab` symbol table class. This is a hash table of `Symbol`s.
3. Write an `Env` class. This is a linked list of `SyTab`s.
4. Create a standard environment consisting of `INTEGER, REAL, CHAR, BOOLEAN, TRUE, FALSE`.
5. Add relevant attributes to the `AST` classes. many of these attributes will be of type `Symbol`, `SyTab`, and `Env`.
6. Write attribute grammar rules for declaration analysis. Start with the rules for *constant declarations* since these zre simple, but once you can handle constants you can handle most everything else! Many of the rules you can extract directly from the handouts. These rules will build up symbol tables for each scope.

7. Write recursive routines to traverse the AST and code the attribute grammar rules.
8. At this point, you're going to need to debug! So, add routines to the `AST` classes to print out all the attributes.
9. Once done with declaration analysis, add rules to construct environments, and pass these down to statements and expressions. Again, debug by printing out the decorated tree, making sure that the environments are correct everywhere!
10. Now, implement statement and expression analysis.
11. Debug, debug, debug!!!

5 Submission and Assessment

- The deadline for this assignment is Wed Oct 28, 23.59. It is worth 10% of your final grade.
- You should submit the assignment electronically to `d2l.arizona.edu`.
- You can work alone or in teams of 2. You must submit a `README` file that lists the members of your team and how much each team member contributed to the assignment.
- If you work in a team you should only submit one copy of the assignment.
- You can download 80 test cases from the class website: `http://www.cs.arizona.edu/~collberg/Teaching/453/2009/Assignments/index.html`. Each will give you one point if you get it right and 0 points if you get it wrong. No partial credits. We won't check for the correctness of line numbers.
- Your electronic submission *must* contain a working `Makefile`, and *all* the files necessary to build the lexer *and* parser. If your program does not compile "out of the box you *will* receive *zero* (0) points. The TA will *not* try to debug your program or your makefile for you!

Don't show your code to anyone outside your team, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.

A Examples

```
PROGRAM T16;
VAR C:CHAR;
BEGIN
C := 'X' * 'Y';
END.
```

$\Longrightarrow$

```
<SEMANTIC_ERROR pos="4" message="Numeric␣type␣expected"/>
<SEMANTIC_ERROR pos="4" message="Numeric␣type␣expected"/>
```

```
PROGRAM T34;
PROCEDURE P(
   I:INTEGER;
   C:CHAR);
BEGIN
END;
BEGIN
P('x',3);
END.
```

$\Longrightarrow$

```
<SEMANTIC_ERROR pos="8" message="Actual/formal␣parameter␣type␣missmatch"/>
<SEMANTIC_ERROR pos="8" message="Actual/formal␣parameter␣type␣missmatch"/>
```

```
PROGRAM T394;
VAR x : REAL;
VAR y : REAL;
VAR z : REAL;
BEGIN
   x := y % z;
END.
```

$\Longrightarrow$

```
<SEMANTIC_ERROR pos="6" message="Integer␣type␣expected"/>
<SEMANTIC_ERROR pos="6" message="Integer␣type␣expected"/>
```

B For honor's section only

Extend the semantic analyser to handle the object oriented parts of LUCA. See me for details about the semantic rules.