

## CSE 6341, Programming project 2

Due Monday, September 28, 11:59 pm (20 points)

The goal of this project is to extend the type checking from Project 1 to a more complex language. The new language has more expressions and statements, as well as nested blocks. The type checker should build a tree of symbol tables, similarly to what was discussed in class (slides 66-72). Note that for this programming project we do not introduce function definitions and function calls, but such an addition would be pretty straightforward. Your implementation will use the supplied code for parsing and AST building.

### Step 1: Set up

```
cd /home/buckeye.8/6341/proj
rm -r p2
wget web.cse.ohio-state.edu/~rountev.1/6341/project/p2.tar.gz
tar -xvzf p2.tar.gz
cd p2
make
./plan t1
```

### Step 2: Understand the AST

The context-free grammar for the targeted language is as follows:

```
<program> ::= <unitList>
<unitList> ::= <unit><unitList> | <unit>
<unit> ::= <decl> | <stmt>
<decl> ::= <varDecl> ; | <varDecl> = <expr> ;
<varDecl> ::= int ident | float ident
<stmt> ::= ident = <expr> ; | print <expr> ;
           | if ( <condExpr> ) <stmt>
           | if ( <condExpr> ) <stmt> else <stmt>
           | while ( <condExpr> ) <stmt>
           | { <unitList> }
<expr> ::= intconst | floatconst | ident
           | <binaryExpr> | - <expr> | ( <expr> ) | readint | readfloat
<binaryExpr> ::= <expr> + <expr> | <expr> - <expr> | <expr> * <expr> | <expr> / <expr>
<condExpr> ::= <expr> <b>< <expr> </b> | <expr> <b>= <expr> </b> | <expr> <b>> <expr> </b> | <expr> <b>=> <expr> </b>
           | <expr> <b>== <expr> </b> | <expr> <b>!= <expr> </b> | <condExpr> <b>&& <condExpr> </b>
           | <condExpr> <b>|| <condExpr> </b> | <b>! <condExpr> </b> | ( <condExpr> )
```

Some new features: (1) if-then, if-then-else, and while-loops; (2) nested blocks **{ <unitList> }**; (3) more general binary arithmetic expressions; (4) unary minus **- <expr>**; (5) expressions to read from UNIX *stdin* an int value (**readint**) or a floating-point value (**readfloat**); (6) general boolean expressions. Read the code in p2/ast to see how the AST nodes are defined. The root of the AST is a node that is an instance of the Program class. It is important to do this reading early and to ask any clarification questions as soon as possible.

### Step 3: Implement the generalized type checking

You need to implement a type checker to check for the following conditions:

1) Any variable appearing in an **<expr>** must have a declaration in some earlier **<decl>**, including declarations in surrounding blocks. For example, the following code is correct

```
int x = readint + 7;
float y = readfloat + 5. + readfloat;
if (x>0) { x = x + 1; int z = x + 2; { int p = z + 3; x = p + 4; } z = z + 5; { int q = x + 6; } }
```

However, if we replaced **z + 3** with **q + 3**, the code would not type check.

2) Each variable can be declared only once inside each **<unitList>**. So, a program of this form is not valid: **int x = ...; float y = ...; int x = ...**. Similarly, **{ int z = ...; { ... } int z = ...; }** is not valid. However, **int z; { float z = ...; { int z = ...; } }** is valid since the different declarations of **z** are not part of the same **<unitList>**.

3) The *innermost surrounding declaration* is used to define the type of an occurrence of a variable. For example, **int z = ...; { float z = ...; { int z = ...; } { float w = z + 1.1; } }** is valid because the occurrence of **z** in **z + 1.1** is matched with the innermost surrounding **float z** declaration. If this declaration were deleted from the program, the top-level declaration **int z** would apply to **z + 1.1** and the program would not type check.

4) In an assignment **ident = <expr> ;** the variable on the left-hand side of the assignment must have a declaration in some earlier **<decl>**, including declarations in surrounding blocks.

5) In an assignment **ident = <expr> ;** or a declaration with initialization **<varDecl> = <expr> ;** the type of the variable on the left-hand side of **=** must be the same as the type of the expression on the right-hand side.

6) Both operands of a binary arithmetic operator **+ - \* /** must be of the same type (either INT or FLOAT). The result of the operation is of that same type.

- 7) The operand of a unity minus operator can be either INT or FLOAT. The result of the operation is that same type.
- 8) The evaluation of a readint expression produced a value of type INT. The evaluation of a readfloat expression produces a value of type FLOAT.
- 9) Both operands of a comparison operator `< <= > >= == !=` must be of the same type (either INT or FLOAT).

If the program violates any of these checks, call `Interpreter.fatalError` with exit code `EXIT_STATIC_CHECKING_ERROR`. The test script will check this exit code, so please make sure your implementation uses it. The text message associated with the error should be something simple that describes which specific check was violated.

A natural way to implement the checking is to add to each class in package `ast` a method `check` which takes as input a reference to some node in the tree of symbol tables. However, feel free to ignore this suggestion completely.

#### Step 4: Testing

Write many test cases and test your checker with them. Submit at least 5 test cases with your submission. The test cases will not affect your score, but I will examine them to see how you approached testing. Put them in the same location as the provided file `t1` and name them `t2`, ...

#### Step 5: Submission

After completing your project, do

```
cd p2
make clean
cd ..
tar -cvzf p2.tar.gz p2
```

Then submit `p2.tar.gz` in Carmen.

#### General rules (copied from the course syllabus)

- 1) Your submissions must be submitted electronically via Carmen by midnight on the due date. The projects must compile and run on **stdlinux**. Some students prefer to implement the projects on a different machine, and then port them to stdlinux. If you decide to use a different machine, it is entirely your responsibility to make the code compile and run correctly on stdlinux before the deadline. In the past many students have tried to port to stdlinux too close to the deadline, leading to last-minute problems and missed deadlines.

- 2) Projects should be done independently. General high-level discussion of projects with other students in the class is allowed, but you must do all design, programming, testing, and debugging independently. Projects that show excessive similarities will be taken as evidence of cheating and dealt with accordingly. Code plagiarism tools may be used to detect cheating. See more details in the Syllabus under "Academic integrity":  
<http://web.cse.ohio-state.edu/~rountev.1/6341/pdf/CSE6341-Syllabus-Au20.pdf>
- 3) The projects are due by 11:59 pm on the due day. No exceptions will be made to this deadline: if you submit at 12:00 am, your submission will be late. Please plan your time carefully and do not submit in the last minute. You can submit up to 24 hours after the deadline; if you do so, your project score will be reduced by 10%. If you submit more than 24 hours after the deadline, the submission will not be accepted, and you will receive zero points for this project.