PA6: Register All

ITP 439

PA2: Semantic Analysis

Introduction

Theme: auto

plemented a parser for the USC language. However, the parser still any invalid programs such that later phases of the compiler may assume that the program is valid.

Some semantic analysis has already been implemented for you – specifically related to functions and arrays. However, you will be responsible for implementing most of the semantic checks. **PA5: Optimization**

c rules you will implement are related to declarations of identifiers as well as type that identifiers are used only after they are declared, you will implement a symbol supports a small number of types, the only type conversions required are from

> versa – all other types are incompatible with each other. USC are simple enough that all semantic checks can occur at the same time the means there is no need for any post-process passes over the AST. You can verify of the program in all the parse functions you wrote in PA1.

errors, you need to set the mCheckSemant member in Parser to true in the ear the top of parse/Parse.cpp. Once you enable semantic errors, the will no longer pass all the tests. For this assignment, the test suite you want to use instead is testSemant.py. To execute this suite, run the following command from the tests directory:

nand line flag, -1, (that's the letter L not a 1), which tells USCC to also output the symbol table information for debug purposes.

To add this argument in Visual Studio Code, you will need to edit your args array to look like this:

for this assignment, you will have more and more tests pass.

As in BA1, you need to make sure your error measures match the expected format, or you will not bass all the less.

Symbol Table

Implementation

iden fier declarations. This means every time an open brace is encounte led, a new scope is ergated and the scope is closed on its matching closing brace. The language also allows for *ghosting* of variables, meaning that a child scope can hide an identifier declared in a parent scope if it uses the same name. However, two variables in the same scope level cannot share the same name.

To account for this scoping, you will implement a scoped symbol table, also known as the sheaf-oftables approace. This type of syribol able is covered in §5 5.1 of Engineering a Compiler (2nd identifier it first checks the current scope before walking the parent hierarchy in an effort to locate the symbol.

The symbol table is declared in parse/Symbols.h and implemented in parse/Symbols.cpp. You will need to implement every function in SymbolTable and its nested class ScopeTable. The only exception is the ScopeTable::emitIR function, which you will add implementation details to in PA3.

The majority of the functions should be self-explanatory both based on the comments and the names/parameters for the functions. However, the constructor for SymbolTable needs to do a few things which will not be obvious:

- 1 Enter the global scope (create the root ScopeTable)
- 2 Create an identifier named @@function and set its type to function
- 3 Create an identifier named @@variable and set its type to integer
- 4 Create an identifier named printf and set its type to function

The cofunction and covariable identifiers are "dummy" identifiers. When parsing any statement or expression that uses an identifier, you will first check whether or not that identifier is declared. If it's not, you will emit a semantic error and set its identifier to point to the corresponding "dummy" one. This way you can avoid getting a chain of several semantic errors for the same statement. The printf identifier is necessary because USC automatically provides support for the printf function from the C standard library. Without the identifier, any calls to printf in USC source code would have an undeclared identifier error.

Integrating the Symbol Table

There are a few things you need to do in order to integrate the symbol table. First, you need to enter/exit scope at the appropriate points. The code for functions is already provided for you, but you need to add the enter/exit hooks in parseCompoundStmt, which is in parse/ParseStmt.cpp. If the isFuncBody parameter is set to true, you don't want to create a scope because a scope was already created when parsing the function node.

Next, you need to implement the Parser::getVariable function in parse/Parse.cpp. This function should first try to get the identifier from the symbol table, and return the identifier if it's found. If it doesn't exist, it should output a semantic error via the reportSemantError function, and then return the identifier for the dummy @@variable. The reportSemantError function optionally takes in a column override as a parameter. This is to

account for the fact that the parse may have advanced a couple of tokens prior to recognizing the semantic error. So you can use this override to ensure the error caret lines up properly.

Because of the way the test cases work, the format of the error messages has to match the reference code. For example, the error messages for an undefined identifier x should be:

Use of undeclared identifier 'x'

You also need to ensure that a declaration statement doesn't try to redeclare a variable in the same scope. This should be done in parseDecl, and you can use isDeclaredInScope to check. For example, a redeclaration error message for identifier x should be:

Invalid redeclaration of identifier 'x'

Type Checking and Conversion

The only types USC allows conversions between are char and int, in both directions. However, the policy for expressions is that they should all be aggressively converted into integer types. Integers are then only converted to characters in the cases where it's necessary, such as assigning an integer to a character variable.

There are then two functions declared in parse/Parse.cpp: charToInt and intToChar. The job of these functions is to create a conversion node if necessary.

To assist with these conversions, there are two expression nodes: ASTToIntExpr and ASTToCharExpr.

For example, if the expression passed to charToInt is already an integer, you can just return the expression directly. Otherwise, if it receives an expression of type char, it will do one of two things:

- If the expression is an ASTConstantExpr, you can simply change its type to int • Otherwise, you need to create an ASTToIntExpr node as a parent to the expr
- The intToChar function is similar, though in the opposite direction. You should also optimize this function such that if the expression in question is an ASTToIntExpr, rather than reconverting it, you can grab the child of the ASTToIntExpr node, which should already be a char.

Integrating intToChar and charToInt A call to charToInt is necessary whenever an identifier is accessed in an expression node. This should

be just in parseIdentFactor, parseIncFactor, and parseDecFactor. For all three of these cases, you can defer conversion until right before returning the expression. The usage of intToChar is a bit more complex - you want to use it in parseAssignStmt, parseDecl, and

parseReturnStmt. However, you first need to check and see if the type of the expression is int and if the type you are expecting is a char. Any other pairs of mismatched types should report a semantic error. For assignments and declarations, the error message format should be: Cannot assign an expression of type fromType to toType

For return statements, the error message format should be:

You can get the name of the type via the getTypeText function.

Expected type nameOfType in return statement

Type Checking for Ops Now you need to add code to check and make sure logical and, logical or, binary math, and binary

comparison ops are all only trying to operate on integers. You don't need to check for characters, because those would have already been converted to integers in child expressions. To help with this, there are four finalizeOp functions at the top of parse/ASTExpr.cpp. These functions just need to set the type of the node to integer, and return true if both lhs and rhs are integers, false otherwise. Once you've implemented these functions, you should hook them up to the appropriate parsing functions. If the finalizeOp function returns false, the error message format should be:

Cannot perform op between type lhsType and rhsType

Return Statements

Now you need to ensure that all functions end with a return statement. The best place to check is in parseCompoundStmt, once you've gotten all the statements. You need to check if the compound

statement is a function body, because only those must end with a return. • If the function is void and it doesn't have a return statement at the end, you need to manually create a void return node and add it (it's necessary to ensure valid code is emitted in PA3).

- Otherwise, if the function is supposed to return a value but doesn't end in a return statement, you should output the following error message:
- USC requires non-void functions to end with a return

Next, you need to check the actual return statements in parseReturnStmt and ensure that if the function is supposed to return a value, it doesn't just have an empty return statement. The error

Invalid empty return in non-void function

Disallowing Reassignment of Arrays

Reassignment of arrays is not allowed

message for this case should say:

The last semantic error to check for is attempting to reassign an array (as a whole, not a subscript). This should be checked in parseAssignStmt, and the error message should be:

You should now pass all of the semantic error test cases.

Testing on GitHub Once you pass all the tests locally, you should push your code to GitHub and manually run the PA2 unit tests. If they don't pass or your code doesn't compile, you should make the necessary fixes and

try again. For this assignment, there are a total of 43 test cases. Each test failure is -2.5 points.

Submitting Your Assignment Once you have a GitHub Actions run you are happy with, to submit the assignment you need to

submit this form on Gradescope. You will have to provide the full URL for the actions run you want us

to grade, and also tell us if you are using any of your four slip days. **DANGER**

Make sure you submit this form, as otherwise we will not know that you have a submission you want us to grade. We only grade based on your GitHub Actions run and code on GitHub, and we will not download everyone's repo. If your code does not compile or you do not submit the form, you will get a 0 on the assignment.

Once you've submitted your assignment, you should make a tag so it's easier to go back to the

Making a Git Tag

git tag pa2 git push origin ——tags

Conclusion

documentation theme for Jekyll.

With the front end complete, uscc will now reject all invalid source programs. In the next assignment, you will write code to convert the AST into a lower-level LLVM IR.

This site is intended for individual educational use only. Redistribution of this content is prohibited without prior approval from the ITP 439 instructors, and may be deemed an academic integrity violation. This site uses Just the Docs, a

specific pa2 submission later, should you need to: