Programming Project:

Compiler construction for MiniJava

Overview

The course project is to implement the MiniJava language. MiniJava is (almost exactly) a subset of Java and the meaning of a MiniJava program is given by its meaning as a Java program.

Compared to full Java, a MiniJava program consists of a single class containing a static `main` method followed by zero or more other classes. There are no other static methods or variables. Classes may extend other classes, and method overriding is included, but method overloading is not. You may assume that there is no predefined `Object` class and that classes defined without an `extends` clause do not implicitly extend `Object`. All classes in a MiniJava program are included in a single source file.

The only types available are `int`, `boolean`, `int[]` and reference types (classes). "`System.out.println(...);`" is a statement and can only print integers - it is not a normal method call, and does not refer to a static method of a class. All methods are value-returning and must end with a single `return` statement.

The only other available statements are `if`, `while`, and assignment. There are other simplifications to keep the project size reasonable.

You should implement full MiniJava as described there, except that, for our project, `/* comments */` are not nested, i.e., you need to implement `/* */` comments, but the first `*/` terminates any open comment regardless of how many `/*` sequences have appeared before it, as in standard Java. You also need to implement `//` comments.

There are two symbols in the grammar that are not otherwise specified. An <IDENTIFIER> is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase. An <INTEGER_LITERAL> is a sequence of decimal digits that denotes an integer value.

Implementation

You are free to use any development environment you wish, but Java is recommended.

Scanner

The purpose of this assignment is to construct a scanner for MiniJava. You will need to examine the MiniJava source grammar to decide which symbols are terminals and which are non-terminals (hint: be sure to include operators, brackets, and other punctuation -- but *not* comments and whitespace -- in the set of terminal symbols).

For the compiler itself you should create an appropriate main program and you will need to create an appropriate set of tokens for MiniJava.

You should create a Java class named `MiniJava` with a `main` method that controls execution of your compiler. This method should examine its arguments (the `String` array parameter that is found in every Java `main` method) to discover compiler options and the name of the file to be compiled. When this method is executed using the command

```
java MiniJava -S filename.java
```

the compiler should open the named input file and read tokens from it by calling the scanner repeatedly until the end of the input file is reached. The tokens should be printed on standard output (Java's `System.out`).

When the compiler (i.e., just the scanner at this point) terminates, it **must** return an "exit" or status code indicating whether any errors were discovered when compiling the input program. In Java the method call `System.exit(`*status*`)` terminates the program with the given *status*. The status value should be 0 (normal termination) if no errors are discovered. If an error is detected, the exit status value should be 1.

Note: Your compiler must read input from the file named on the `java` command, so you will need to include appropriate code in your `MiniJava` main program to open that file and prepare it for reading.

To test your scanner, you should create a variety of input files, including some that contain legal MiniJava programs and others that contain random input. You should save these test programs in your source repository in some appropriate place. Be sure your scanner does something reasonable if it encounters junk in the input file. (Crashing, halting immediately on the first error, or infinite looping is not reasonable; complaining, skipping the junk, and moving on is.) Remember, it is up to the parser to decide if the tokens in the input make up a well-formed MiniJava program; the scanner's job is simply to deliver the next token whenever it is called.

Parser +AST

For this part of the project, construct a parser (recognizer) and build abstract syntax trees (ASTs) for MiniJava. You should use any parser generator tool to interface with your scanner. Be sure that your parser and scanner together can successfully parse legal MiniJava programs.

The semantic actions in the parser should create an Abstract Syntax Tree (AST) representation of the parsed program. For now, just build the AST. Semantic analysis and type checking will be included in the next part of the project.

In addition to building the AST, you should provide an implementation of the Visitor pattern to print a nicely indented representation of the tree on standard output. You should also use the `PrettyPrintVisitor` to print a "decompiled" version of the AST as a MiniJava source program when requested. These output formats are different and more details are given below.

Your m`ain` program should parse the MiniJava program in the named input file and print an abstract representation of the AST on standard output. Similarly, it should parse the MiniJava program in the named input file and print on standard output a decompiled version of the AST ("prettyprint") in a format that is a legal Java program that could be processed by any Java compiler.

As with the scanner, when your compiler terminates it should return an exit or status code of 0 if no errors were detected while compiling (scanning and parsing) the input program. It should return an exit code of 1 if any errors were detected.

To make this a bit more concrete, suppose we use this input file:

```
class Foo {
      public static void main(String[] args) {
    System.out.println(5 + 4 * 3 - 1);
}
}

class Bar extends Nothing { public int bar(Baz f, Bar b) {
  System.out.println(new Thing().method());
    return 5;
  }
      }
```

The abstract AST printout should resemble this output:

```
Program
  MainClass Foo (line 1)
    Print (line 3)
      ((5 + (4 * 3)) - 1)
  Class Bar extends Nothing (line 7)
```

```
    MethodDecl bar (line 7)
      returns int
      parameters:
        Baz f
        Bar b
      Print (line 8)
        new Thing().method()
      Return 5 (line 9)
```

while `MiniJava` should prettyprint the file and produce something resembling this output.

```
class Foo {
  public static void main(String[] args) {
    System.out.println(((5 + (4 * 3)) - 1));
  }
}
class Bar extends Nothing {
  public int bar(Baz f, Bar b) {
    System.out.println(new Thing().method());
    return 5;
  }
}
```

Your actual output may differ slightly, but it should be similar. Note that Foo.java is clearly an illegal Java program, but it is syntactically correct, so you should be able to parse it and build an AST.

If the input program contains syntax errors, it is up to you how you handle the situation. You can simply report the error and quit without producing the AST, or, if you wish, you can try to do better than that.

Feel free to experiment with language extensions (additional Java constructs not included in MiniJava) or syntactic error recovery if you wish, but be sure to get the basic parser/AST/visitors working first.

Details

You may need to massage the MiniJava grammar to make it LALR(1) to produce a parser. Please keep track of the changes you make and turn in a description of them with this part of the project.

Take advantage of precedence and associativity declarations in the parser specification to keep the overall size of the parser grammar small. In particular, `exp ::= exp op exp` productions along with precedence and associativity declarations for various operators will shorten the specification considerably compared to a grammar that encodes that information in separate productions.

Hint/warning: Be sure to test precedence and associativity carefully, and not just in the obvious cases involving operators like `+` and `*`. You should be sure that things like variables, array element references, parenthesized subexpressions, and method calls in expressions interact correctly with other operations and that the resulting ASTs have the right structure (your AST print visitor can be very helpful here).

Your grammar should not contain any reduce-reduce conflicts, and should have as few shift-reduce conflicts as possible. You should describe any remaining shift-reduce conflicts in your writeup and explain how they are resolved and why this resolution is appropriate.

After the parser is generated, a MiniJava compiler parses its input by calling the `parse()` method. Create your `debug_parse()` and if it is called the parser should print a detailed trace of all of the shift and reduce actions performed as it parses the input. That information can be very useful when trying to figure out parser problems.

Once you have the parsing rules in place and have sorted out any grammar issues, add semantic actions to your parser to create an Abstract Syntax Tree (AST) and add Visitor code to print a nicely indented representation of the AST on standard output. Also add code to use the supplied `PrettyPrintVisitor` class to print the decompiled version of the AST when requested.

The AST should represent the abstract structure of the code, not the concrete syntax. Each node in the tree should be an instance of a Java class corresponding to a production in an abstract grammar. The abstract grammar should be as simple as possible, but no simpler. Nodes for nonterminals should contain references to nodes representing non-trivial component parts of the nonterminal, i.e., the important items on the right-hand side of an abstract grammar rule.

Trivial things like semicolons, braces, and parentheses should not be represented in the AST, and chain productions (like *Expression -> Term -> Factor -> Identifier*) should not be included unless they convey useful semantic information. The classes defining the different abstract syntax tree nodes should take advantage of inheritance to build a strongly-typed, self-describing tree.

The output of the new AST Visitor should be a readable representation of the *abstract* tree, *not* a "de-compiled" version of the program, and, in particular, the output will not be a syntactically legal Java program that could then be fed back into a Java compiler. Each node in the tree should normally begin on a separate line, and children of a node should be indented below it to show the nesting structure. The output should include source line numbers for major constructs in the tree (certainly for individual statements and for things like class, method, and instance variable declarations, but not necessarily for every minor node in, for example, expressions).

The abstract tree printout should not include syntactic noise from the original program like curly braces (`{}`), semicolons, and other punctuation that is not retained in the AST, although you should use reasonable punctuation (indentation, whitespace, parentheses, commas, etc.) in your output to make things readable. Although most tree nodes should occupy a separate line in the output, you can, if you wish, print things like expressions on fewer lines to reduce the vertical space used, as long as your output clearly shows the AST structure of the expression (perhaps by adding parentheses to show nesting if line breaks and indenting are not used).

You should test your parser, semantic actions, and printing visitor by processing several MiniJava programs, both correct ones and ones with syntax errors.

What to Hand In

The main things we'll be looking at in this phase of the project are the following:

- The grammar specification, including semantic actions, for your parser.
- Source files containing definitions of the AST node classes and visitors.
- Printed AST representations produced by print visitors.
- A description of any changes you made to the grammar, and a list of shift-reduce conflicts remaining in the grammar (if any) with an explanation of why these are not a problem.
- If you include any error handling or extra features in your parser, or add any language extensions, include a brief description of these.

Semantics +Types

Add static semantics checking to your compiler and add code to print the resulting symbol tables so the semantics and type information can be verified. In particular, you should check for the following:

- Add global symbol table(s) storing information about classes and their members, and local symbol tables for each method to store information about parameters and local variables. (Hint: recall that in Java and MiniJava a single identifier can be used as both an instance variable and a method name in the same class. Be sure you handle this case either with multiple symbol tables or some other appropriate mechanism.)
- Calculate and store type information for classes, class members, parameters, and variables.
- Calculate type information for expressions and other appropriate parts of the abstract syntax.
- Add error checking to verify at least the following properties:
    i. Every name is properly declared.
    ii. Components of expressions have appropriate types (e.g., `+` is only applied to values of type `int`, `&&` is only applied to values of type `boolean`, the expression in parentheses in an `if` or `while` statement is `boolean`, etc.).
    iii. If a method is selected from a value with a reference type, then that name is defined as a member of that type.
    iv. Methods are called with the correct number of arguments.
    v. In assignment statements and method call parameter lists, the values being assigned have appropriate types (i.e., the value either has the same type as the variable or is a subclass of the variable's class).
    vi. If a method in a subclass overrides one in any of its superclasses, the overriding method has the same parameter list as the original method, and the result type of

the method is the same as the result type of the original one, or a subclass if the original result type is a reference type.

Feel free to add additional checks, but you should try to get this much done. It may be possible to do some of these checks on the fly while building symbol tables and type information, or it may require an additional visitor pass.

- Print suitable messages describing any errors detected. It's fine to suppress useless error messages - for instance, feel free to complain only once about an undeclared variable instead of repeating the message each time the variable is used in the code.

It is required to parse the MiniJava program in the named input file, perform semantic checks as described above, and print the contents of the compiler symbol tables.

As with previous parts of the compiler project, the compiler main method should return an exit or status code of 1 if any errors are detected in the source program being compiled (including errors detected by the scanner or parser, as well as semantics and type checking). If no errors are found, the compiler should terminate with a result code of 0.

Details and Suggestions

It's probably easiest to collect the type information in multiple passes over the AST. An initial pass should collect information about classes and fields (both data and methods), and build the global symbol tables. A later pass would then analyze method bodies, build the local symbol tables, and perform type and other error checking. You might find it more convenient to break this down into more passes, each of which does fewer things, particularly for the initial pass where it might be easier to build a global symbol table of class names before processing individual classes to build class symbol tables with information about variables, methods, and their types.

You should add appropriate fields in some or all AST nodes to store references to type and other information as necessary. But remember that you should have a separate data abstraction (ADT) to represent type information used for semantics checking in the compiler, and not confuse this information with the source program type declarations in the AST.

It can be useful to include a few auxiliary methods that perform common operations on types. Possibilities include a method that returns true if two types are the same, and a method that returns true if a value of one type is assignable to another. Also possibly useful: a method that tries to add an entry to a symbol table and reports an error if the name is already declared, and another that looks up an identifier and reports an error if it is not found (and maybe adds it to the symbol table with an "undefined" type, which can be used to suppress additional redundant error messages about the same identifier).

You should test your compiler by processing several MiniJava programs, both correct ones and ones with errors. Be sure to check some examples that are syntactically legal (i.e., can be parsed with no errors) but that contain semantic errors.

What to Hand In

For this phase of the project we will be looking to see if your compiler properly performs at least the semantics checks listed in the Overview section above, and can print the requested symbol tables and information in a reasonable format. We also will check whether your compiler can handle MiniJava programs containing errors as well as ones that are legal.

Code Generation

We suggest that you use the simple code generation strategy outlined in class to be sure you get running code on time, although you are free to do something different (i.e., better) if you have time. Whatever strategy you use, remember that simple, correct, and working is better than clever, complex, and not done.

We also strongly suggest thorough testing after you implement each part of the code generator. Debugging of code generators can be difficult, and you will make your life easier if you find bugs early, before your generator is too complex. Using a test-driven development approach has also been effective in the past for groups that have tried it -- i.e., writing tests for particular language features prior to writing the code generation that implements them.

If translation is successful, the compiler should terminate with `System.exit(0)`. If any errors are detected in the input program, including syntax, static semantics or type-checking errors, the compiler should terminate using `System.exit(1)`. If errors are detected, the compiler does not need to produce any assembly language code, and, in fact, should probably exit without attempting to do so. (The code generator should be able to assume it is translating a correct MiniJava program and should not need to include error checks or special cases to deal with incorrect input source code.)

The output program should be a correct translation of the original MiniJava program and the generated code should not produce runtime errors like segfaults, to the extent this is reasonable. In particular, if a MiniJava program attempts to access an array element with an illegal (out of bounds) subscript, execution should be terminated with an appropriate error message. It is up to you whether the message contains the source line number of the error, although it would be useful to include this. You do not need to generate code to check all object references for possible null pointers or deal with other situations where it would be unreasonable to try to detect problems.

Implementation Strategy

Code generation incorporates many more-or-less independent tasks. One of the first things to do is figure out what to implement first, what to put off, and how to test your code as you go. The following sections outline one reasonable way to break the job down into smaller parts. We suggest that you tackle the job in roughly this order so you can get a small program compiled and running

quickly, and add to the compiler incrementally until you're done. Your experience implementing the first parts of the code generator also should give you insights that will ease implementation of the rest.

Integer Expressions & System.out.println

First add code generation for basic arithmetic expressions including only integer constants, `+`, `-`, `*` and parentheses. You will also need to generate the basic function prologue and exit code for the MiniJava `main` method.

Object Creation and Method Calls

Next, try implementing objects with methods, but without instance variables, method parameters, or local variables. This includes:

- Operator new (i.e., allocate an object with a method table pointer, but no fields)
- Generation of method tables for simple classes that don't extend other classes
- Methods with no parameters or local variables.

Once you've gotten this far, you should be able to run programs that create objects and call their methods. These methods can contain System.out.println statements to verify that objects are created and that evaluation and printing of arithmetic expressions works in this context.

Variables, Parameters, & Assignment

Next try adding:

- Integer parameters and variables in methods, including assigning stack frame locations for variables.
- Parameters and variables in expressions
- Assignment statements involving parameters and local variables

Suggestion: Some of the complexity dealing with methods is handling registers during method calls. It can help to develop and test this incrementally -- first a single, simple function argument, then multiple arguments, then arguments that require evaluation of nested method calls.

Control Flow

This includes:

- While loops
- If statements
- Boolean expressions, but only in the context of controlling conditional statements and loops.

Classes and Instance Variables

Add the remaining code for classes that don't extend other classes, including calculating object sizes and assigning offsets to instance variables, and using instance variables in expressions and as the target of assignments. At this point, you should be able to compile and execute substantial programs.

Extended Classes

The main issue here is generating the right object layouts and method tables for extended classes, including handling method overriding properly. Once you've done that, dynamic dispatching of method calls should work, and you will have almost all of MiniJava working.

Arrays

We suggest you leave this until late in the project, since you can get most everything else working without arrays.

The Rest

Whatever is left, including items like storable Boolean values, which are not essential to the rest of the project, and any extensions you've added to MiniJava.

C Bootstrap

As discussed in class, the easiest way to run the compiled code is to call it from a trivial C program. That ensures that the stack is properly set up when the compiled code begins execution, and provides a convenient place to put other functions that provide an interface between the compiled code and the outside world.

Preparing and Executing x86-64 Code with gcc

Your compiler should produce output containing x86-64 assembly language code suitable as input to the GNU assembler `as`. You can compile and execute your generated code and the bootstrap program using `gcc` at the x86-64 instruction level.

Extensions

The basic project requirements are small enough to be tractable in a one-quarter course project, but include enough to cover the core ideas of compiling object-oriented (as well as procedural) languages.

If you are feeling ambitious and have the time, you are invited to add additional parts of Java to the language. Here are a few suggestions:

- dd additional arithmetic and relational operators for integers that are not included in the basic MiniJava specification.
- Add `null` as a constant expression and include `==` and `!=` for object reference types.
- Allow `return` statements anywhere in a method.
- Allow calls of local methods without the explicit use of `this`.
- Relax the ordering of declarations so that variables can be declared anywhere in a method body.
- Allow initialization of variables in declarations.
- Add `void` methods, a `return` statement with no expression, and appropriate type checking rules.
- Support `public` and `private` declarations on both methods and instance variables, and check to ensure that access restrictions are not violated.
- Add a top-level `Object` class that is implicitly the superclass of any class that does not explicitly `extend` another class.
- Add `String` literals, extend `System.out.println` to support `String`s and overload the `+` operator for `String` values.
- Add `double` as a numeric type, overload `System.out.println` to print `double`s, and implement arithmetic operations for `double`, possibly including mixed-mode integer and floating-point arithmetic.

More Sophisticated, but very interesting

- Support `instanceof` and type casts (which can require a runtime `instanceof` check). (This actually is surprisingly easy to do.)
- Support `super.` as a prefix in method calls.
- Add constructor declarations with optional parameters. To make this useful, you probably want to include the use of `super(...)` at the beginning of the body of a constructor.
- Support arrays of objects (no harder to implement than arrays of `int`s, except the type checking rules are more interesting).

Of the suggested extensions, adding `instanceof`, type casts, and `super.` are particularly instructive.

Some amount of extra credit will be awarded to projects that go beyond the basic requirements.

BNF Grammar for MiniJava

Goal ::= MainClass (ClassDeclaration)*

MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")" "{" Statement "}" "}"

ClassDeclaration ::= "class" Identifier ("extends" Identifier)? "{" (VarDeclaration)* (MethodDeclaration)* "}"

VarDeclaration ::= Type Identifier ";"

MethodDeclaration ::= "public" Type Identifier "(" (Type Identifier ("," Type Identifier)*)? ")" "{" (VarDeclaration)* (Statement)* "return" Expression ";" "}"

Type ::= "int" "[" "]"
| "boolean"
| "int"
| Identifier

Statement ::= "{" (Statement)* "}"
| "if" "(" Expression ")" Statement "else" Statement
| "while" "(" Expression ")" Statement
| "System.out.println" "(" Expression ")" ";"
| Identifier "=" Expression ";"
| Identifier "[" Expression "]" "=" Expression ";"

Expression ::= Expression ("&&" | "<" | "+" | "-" | "*") Expression
| Expression "[" Expression "]"
| Expression "." "length"
| Expression "." Identifier "(" (Expression ("," Expression)*)? ")"
| <INTEGER_LITERAL>
| "true"
| "false"
| Identifier
| "this"
| "new" "int" "[" Expression "]"
| "new" Identifier "(" ")"
| "!" Expression
| "(" Expression ")"

Identifier ::= <IDENTIFIER>