Compiler Construction - Assignment 2

LIACS, Leiden University

Fall 2008

Introduction

For the remainder of the practical part of the Compiler Construction course, we will study a subset Pascal compiler. In this assignment, you will add syntax tree construction to a framework for the subset Pascal compiler. A syntax tree is the first internal representation of an input program. It is constructed during the parsing process. Next, you also have to implement several semantic checks. It is often convenient to do these checks during syntax tree construction.

Framework

In your CVS repository, a module named "`assignment2`" is available. This module contains a `bison`-framework for a subset Pascal compiler. A makefile is provided; after performing a checkout of the CVS module, first execute `make first` once. In order to build the entire compiler, simply execute `make`.

Data structures

Several data structures that are needed during the syntax tree construction phase are already provided. In the next paragraphs, we briefly describe them. Note that you are not allowed to make any changes to these data structures. Futhermore, you should avoid triggering warning or error messages from the different object methods.

Node

The `SyntaxTree` is constructed out of `Node`s. Each node has a general type (e.g., `NODE_WHILE`, `NODE_ADD`, `NODE_FUNCTIONCALL`) and a return type (e.g., `RT_INT`, `RT_VOID`). The node types and return types are listed in the source file `types.h`. For each node type, a brief usage convention is provided in the `types.h` file. You have to stick to this convention.

Several node variants are available: parent nodes with one (`Node_Unary`) and two (`Node_Binary`) child nodes and leaf nodes for integer values (`Node_Integer`), real values (`Node_Real`) and symbols (`Node_Symbol`).

SyntaxTree

The `SyntaxTree` data structure stores trees of `Node`s representing the input program. For each subprogram (i.e., a procedure or function), a tree consisting of nodes has to be generated. This tree is then inserted into the global syntax tree using the `AddSubprogram` method. The main program body of an input program is stored using the `SetProgramBody` method. Various functions for creating nodes and leafs are available (`CreateParentNode`, `CreateLeaf`). You should avoid the creation of cycles in the tree.

Symbol

The `Symbol` data structure stores all relevant information of an object in the input program. This includes the object name, the line on which it was declared, the object type (e.g., a function or variable) and the return type. For variables, the return type corresponds to the type of the variable. For procedures, it should be set to `RT_VOID`. For functions, it corresponds to the return type of the function.

Scope

In order to guarantee correct referencing of variables etc., the compiler needs to keep track of scope information. In the framework, this is done using `Scope` objects. For each scope region of the input program, a new `Scope` object has to be created. This `Scope` object then stores all symbols found in this scope level.

SymbolTable

The symbol table stores all symbols of the input program and provides methods to insert new symbols and lookup previously inserted symbols. It actually consists of a set of `Scope` objects, which in turn store the symbols. The symbol table of this framework has been simplified, since the input language only distuingishes between two different scope levels: a variable is declared either in the main scope, or in the scope of a subprogram. This means you do not have to crawl through a hierarchy of scopes in order to find a certain symbol. Instead, a call to `GetSymbol()` will return the corresponding symbol immediately, provided that the symbol table has been filled correctly.

The Subset Pascal Language

In this section, we briefly describe some of the concepts of the input language your compiler has to support. At a high level, a subset Pascal program can be divided into four different parts:

`program` name`;`
Global variable declarations (optional)
Subprogram declarations and definitions (optional)

`begin`
  Program body
`end.`

The first part defines the program name. This is followed by a list of global variable declarations. If a program does not contain global variables, then this section is left empty. Next, the subprogram declarations and definitions are specified. Two different types of subprograms exist: functions and procedures. A function is a callable subprogram that returns a value, just like a C function. A procedure is a callable subprogram that does *not* return a value, similar to a `void` function in C. If a program does not contain any subprograms, then this section is left empty. Finally, the program body is listed, which is comparable to the `main` function of C-like languages.

An example of a subset Pascal program is given below. This program computes the factorial of 5 using recursive calls and places the result in a global variable.

```
 1  program factorial;
 2
 3  var
 4    result: integer;
 5
 6  function fac(a: integer): integer;
 7  begin
 8    if (a = 0) then
 9      fac := 1
10    else
11      fac := a*fac(a- 1)
12  end;
13
14  begin
15    result := fac(5)
16  end.
```

The global variable `result` is declared on line 4. This variable is accessible in the body of both the program and the subprogram(s). The function `fac` is declared on line 6 and its body is defined on lines 7-12. Note that the language does not offer an explicit return statement. Instead, the return value of a function is assigned to a local variable that carries the function name itself, as can be seen on lines 9 & 11. This special return variable can only be used in the left hand side of an assignment. Finally, on line 15, the result of a call to `fac` with a parameter value of 5, is placed into the global `result` variable.

Finally, we list some possible pitfalls of the language:

- The assignment operator is `:=`, instead of a single equals sign.
- Use semicolons (`;`) to separate different statements. However, the last statement of a (sub)program body should *not* be followed by a semicolon.
- The relational operator to test for equality is `=` (a single equals sign). The relational operator to test for inequality is `<>`.

For more detailed information about the constructs of the input language, we advise you to take a look at the various programs available in the `tests/` directory.

Assignment

What you have to deliver: a compiler which can parse a given code file, construct a syntax tree and determine if the code file is syntactically and semantically correct. You should also provide clear documentation about your implementation decisions. This documentation should be placed in your CVS repository in the form of a file called `DOCUMENTATION.TXT`.

This assignment consists of the implementation of the lexical and syntax analyzer, and the construction of the abstract syntax tree. The grammar of the language you have to support is already given in the `comp.y` file. You are not allowed to modify the grammar. Instead, this file has to be augmented with calls to symbol table access routines and syntax tree construction routines. Your program should read a program and output the symbol table and the syntax tree.

Examples of semantic checks to be performed by your compiler are:

- Are all variables declared?
- Is the program correct with regard to the type system?
- Are the number and types of parameters in calls correct?
- Give a warning in case of a coercion.

Because the compiler implements a subset of Pascal, the semantics of our language is also a subset of the semantics of Pascal. It is therefore up to you what the compiler should accept and what not. However, all semantic tests in the `tests` directory should be covered.

In our Pascal-like language, integers can be coerced into reals. In more complex languages, many more types of coercion are possible, and it becomes unmanageable to explicitly check for them anywhere they may occur. The scalable solution is to have a separate function for coercion. You must implement a coercion system for this simple compiler too.

Output all warnings and errors to `stderr`. Make sure that warning generation can be turned off, preferably using a command line option. Report the number of errors and warnings at the end; do not stop processing after the first error and avoid repeated error messages for the same problem. Generating only one error for each undeclared identifier can be done by inserting this identifier in the symbol table with type `error`, which must be coercible to any other type. Let your compiler exit with exit code 1 if an error in the source file has been found. Otherwise, exit with exit code 0.

Note that it is not necessary to implement semantic checks after construction of the syntax tree. For most checks, it is easier to perform them while constructing the syntax tree.

Some other notes:

- We only allow "boolean-like" expressions as a condition in an if- or while-statement. Only simple boolean-like expressions have to be supported, that is, expressions with only one relational operator.
- The input and output will be implemented by the following procedures and functions:
  - `function readinteger():integer`
  - `function readreal():real`
  - `procedure writeinteger(i:integer)`
  - `procedure writereal(r:real)`

  This means you have to predefine these functions in your symbol table.
- In this and following assignments, we use call-by-value parameter passing.
- You do not have to perform any optimizations for this assignment.

Submission & Grading

Submit your work using CVS. Do not send anything by email. Make sure the latest version of your work has been committed to the CVS repository. Then, tag that version using the command:

```
cvs tag DEADLINE2
```

This tag command has to be issued *before* October 23rd, 2008 at 23:59.

For this assignment, 0-10 points can be obtained, which account for 25% of your final grade. If you do not submit your work in time, it will not be graded. The results, once available, can be found on BlackBoard. Your grade for the assignment will not only depend on the functionality of your compiler, but also on the understandability of your code and the documentation of the design decisions. Documentation should be submitted in English only.

Assistance will be given by Michiel Helvensteijn in room 306/308. A schedule for this can be found on BlackBoard. Additionally, you can go to Sven van Haastregt in room 1.22.