COMP2010 – Compilers Coursework: Language Analyser

Submission Deadlines

Part I: Wednesday 22nd February 2012 @ 12 noon

Part II: Wednesday 21st March 2012 @ 12 noon

Goal

The goal of the 2010 Compiler coursework is to build a language analyser for a given programming language. In particular, the analyser should cover scanning (lexical analyses), parsing (syntax analyses) and scope & type checking (semantic analyses).

The analyser should be built in three steps:

1. First, use JLex and CUP to automatically generate code for your scanner and parser. This broadly requires writing regular expressions covering all legal words in the language (<filename>.lex), and a context free grammar describing its rules (<filename>.cup).
2. Define the data structures you need to build the Abstract Syntax Tree for syntactically correct programs, and extend your CUP specification to enable the automatic creation of the AST for a given input program via CUP semantic actions.
3. Implement type and scope checking by means of one or more visits to the AST, using the Visitor methodology.

A single mark will be given to the whole coursework, and that will count 20% towards your final mark for the COMP2010 module. However, coursework submission has been divided in two steps: a first submission on Wednesday 22nd February 2012 @ 12 noon, covering step 1 above (JLex/CUP specifications), and a second submission on Wednesday 21st March 2012 @ 12 noon, covering steps 2 and 3 above (AST creation and semantic analysis). Detailed submission instructions are given at the end of the document.

Language P Description

The language for which you have to build a parser is called P. A P program consists of a *declaration* part and a *main* program. In the declaration part, global variables, functions and new data types are defined; this part may also be empty. The execution of a P program starts from the main part. Below is an overview of the characteristics of the language.

Comments: text enclosed within `/* ... */`. They may span multiple lines.

Identifiers: start with a letter, followed by an arbitrary number of letters and digits. Identifiers are case-sensitive.

Characters: a single letter or digit wrapped around ` '. Its type is `char`.

Boolean constants: `true`, `false`. They are of type `bool`.

Numbers: can be integers (type `int`) and float (type `float`). Examples of integers: 1, 1234, … Examples of float: 0.1, 3.14, … (for simplicity, you may assume them to be positive).

Sequences: can be strings (type `str`) of length 0 (i.e., "") or more; lists (type `list`) of length 0 (i.e., `[]`) or more (e.g., `[1,2,3]`); and tuples (type `tuple`), of length 0 (i.e., `[||]`) or more (e.g., `[|1,"h",3.6|]`). Strings are sequences of characters only. Lists may contain elements of any type; however, all elements within a list must be of the same type (e.g., all integers, all strings, etc.). Tuples are similar to lists, but different types may be stored within the same tuple. For all sequences, indexing is defined as *id*`[`*index*`]`, it starts at 0 and finishes at `len(`*id*`)-1`, where `len(`id`)` returns an integer indicating the length of sequence *id*. Example: given tuple `t=[|1,"a",5.2,7,[9]|]`, then `t[2]=5.2`, while `t[4]=[9]`; given string `s="hello world"`, then `s[len(s)-1]="d"` (where `len(s)=11`).

Basic Types and Operators

Basic Data Types: `bool`, `int`, `float`, `char`
Basic Data Structures: `str`, `list`, `tuple`
Boolean Operators (defined on type `bool`): `!` (not), `||` (or), `&&` (and)
Numeric Operators (defined on type `int` and `float`): `+ - * / ^`
Sequence Operators (defined on type `list`, `str`, `tuple`): `in`, `not in`, `::` (concatenation) , *s*`[`*i*`]`, `len(`*s*`)`
Comparisons (defined on all basic types): `<, <=, > , >=, ==, !=`

New Data Type Definition

`tdef` *type_id* `:` *declaration* `;`

A declaration is a comma-separated list of fields of the form *id* `:` *type*. Once a new data type has been defined, it can be used as name type in subsequent declarations. Examples:

```
tdef person : name:str, surname: str, age:int;
tdef family : mother: person, father: person; children:list;
```

Variable Declarations

id `:` *type* `;`
id `:` *type* `=` *init* `;`

Variables may or may not be initialised at the time of declaration. For newly defined data types, initialisation consists of a sequence of comma-separated values, each of which is assigned to the data type fields in the order of declaration. Examples:

```
a : tuple = [|1, "two", [1,2]|] ;
b : int = 10;
c : str = "hello world!";
d : person = "Licia", "Capra", 30;
e : char = `a';
f : list = [`a', `b', `c', `d'];
```

Function Definition

`fdef` *functionId* `(`*formal parameter list*`)` `:` *outType* *body*

The formal parameter list is comma separated. Each parameter is defined as *id*`:`*type*. The outType is the returned type of the function. The keyword `void` is used to indicate no return value. The body is enclosed between `{ }` and consists of local variable declarations followed by statements. For functions, the body ends with a return statement; this can be omitted if the return type is `void`. The name of the function (*functionId*) follows the same lexical rules of identifiers. Examples:

```
fdef reverse (l:list): list {
      l2: list=[];
      i : int = 0;
      while (i < len(l)) do {
                l2 = l[i] :: l2;
                i = i+1;
      }
      return l2;
}
```

Statements

Assignment: *var* `=` *expression* `;`
Function call: *functionId* `(` *actual parameter list* `);`
Control Flow: `if (`*expression*`)` *body* `else` *body*
`while (`*expression*`) do` *body*
`repeat` *body* `until (`*expression*`);`
`return` *expression;*

In the definitions above, *var* indicates a variable. The actual parameter list is a comma-separated list of parameters, where each parameter is an arbitrary complex expression. Body consists of local variable declarations (if any), followed by statements, enclosed between `{` and `}`. All statements (apart from `if-else` and `while`) terminate with a semicolon. The return statement is the last statement in the function body; the return expression is optional.

Expressions

Expressions can be formed starting from basic type values and variables, by applying the previously defined operators. Parenthesis can be used to enforce precedence and to disambiguate. For new data type definitions, *id.field* is an expression; also, a function call without the semicolon at the end is an expression. Examples:

`p.age + 10` (where `p: person;` has been declared)
`b - sum (10,c) == 30`
`l1 :: l2 :: [a,b]`

Main Program

The second part of the program contains the main program. This is where the execution of your program starts. It consists of a block, that is, a (optional) list of variable declarations followed by a list of statements, all enclosed between `{ }` . Example:

```
{
  a: list = [1,2,3];
  b: list;
  b = revert (a);
}
```

Part I - Detailed Requirements & Submission Instructions

The specific requirements for your scanner are the following:

- Use JLex (or JFlex) to automatically generate a scanner for the P language.
- Make use of macro definitions where necessary. Choose meaningful token type names to make your specification readable and understandable.
- White spaces and comments should be ignored.
- If an illegal character is found, the scanner should report an error, specifying what portion of the input program caused the error, and at what line this text is located. The scanner should then proceed with the next token type.

The specific requirements for your parser are the following:

- Use CUP to automatically produce a parser for the P language.
- Resolve ambiguities in expressions using the precedence and associativity rules.
- If an error is encountered, the parser should print what rule has been violated, then read past it and continue parsing. If the program is syntactically correct, the parser should print a `parsing successful' message

Your scanner and parser must work together.

Once the scanner and parser have been successfully produced using JLex(JFlex)/CUP, write a Test.java class similar to the one seen during the lectures, to test your code on the test files provided on the course webpage.

Each group should submit to the departmental office a **printed** version of:

- The JLex (JFlex) specification <filename>.lex
- The CUP specification <filename>.cup
- Any other class you have defined [if any]
- CLEAR instructions on how to run the code

A printed copy of the cover sheet available at http://www.cs.ucl.ac.uk/teaching/cwsheet.htm must be filled in, signed and attached to the coursework you submit. Read the plagiarism statement carefully before you sign.

Each group should also upload onto Moodle an **electronic** version of:

- The JLex (JFlex) specification <filename>.lex
- The CUP specification <filename>.cup
- The Test.java file
- Any other class you have defined [if any]

Only one submission per group is necessary.

The deadline for completion of this part of the coursework is Wednesday 22nd February 2012 at 12 noon. Any coursework handed in later than *2 working days* after the deadline will automatically receive a zero mark.

Part II - Detailed Requirements & Submission Instructions

For this part of the coursework, you may either work on the parser you created for Part I, or use the one provided (available on the Moodle course webpage from 27/02/2012).

Creation of the AST

First, carefully design the data structures you will use to represent the parsed program. Second, use CUP semantic actions to build the AST bottom-up. The parser should return the root of the AST if the program complies with the rules of the grammar, or "null" otherwise. Write an implementation of the Visitor pattern that traverses your AST and that (approximately) reproduces in output the input program. Write also a Test.java class similar to the one seen during the lectures, to test your code on the four test files available through the course webpage.

Scope & Type Checking

The last step is to implement a semantic analyser that visits the generated AST, using the visitor methodology seen during the lectures, to perform scope and type checks. The specific scope and type check rules of the language are not provided; as part of this coursework, you have to come up with a **PLAUSIBLE** set of rules, and implement them. Your work will be judged based on BOTH the set of rules you establish, AND on your ability to implement them correctly. Examples of plausible scope rules:

1. Each identifier must be declared in an enclosing scope prior to its use
2. It is not possible to declare identifiers of the same name and type in an overlapping scope

Examples of plausible type rules:

1. Assignments are legal only if the identifier on the left-hand side and the expression on the right-hand side have the same type; it is also possible to assign an `int` to a `double`, but not vice-versa
2. The test expression of `if` and `repeat` statements must be of type `bool`
3. The number and type of actual parameters of methods must match the number and type of the declared formal parameters
4. …

Once you have decided what semantic rules the P language should comply with, implement one or more visitors (as requested by your rules) that walk through the AST you have built during parsing, to check that these rules are respected. If a rule is violated, a semantic error should be reported, specifying what rule has been violated. Write also a Test.java class similar to the one seen during the lectures, to test your code. Produce a suite of test files to demonstrate the capability of your analyser to report errors upon violation of your rules.

Each group should submit to the departmental office a **printed** version of:

- A detailed UML class diagram describing the classes you used to build the AST and their relationship (you may wish to add a 1-page document describing your design); there is no need to print the java code declaring such classes;
- A 2-page document describing, in details, the scope and type check rules you have decided to check;
- The visitor class(es) you have implemented to check the above rules ;
- Test files you have created in order to prove the ability of your semantic analyser to deal with the semantic rules you have defined. For each test file, show the output produced by your semantic analyser; add comments about the errors reported (specify, in particular, what semantic rule was violated and why);
- The JLex (JFlex) specification <filename>.lex and CUP specification <filename>.cup you have been working with
- CLEAR instructions on how to run your whole code (scanner, parser and semantic analyser together).

A printed copy of the cover sheet available at http://www.cs.ucl.ac.uk/teaching/cwsheet.htm must be filled in, signed and attached to the coursework you submit. Read the plagiarism statement carefully before you sign.

Each group should submit also upload onto Moodle an **electronic** version of:

- All the code you used to implement the parser (i.e., JLex specification, CUP specification, all java classes used to build the AST);
- All visitor class(es) you have implemented (to reproduce the input program, and to check your scope and type rules);
- All test files you have created in order to prove the ability of your semantic analyser to deal with the semantic rules you have defined;
- Your Test.java file.

Only one submission per group is necessary.

The deadline for completion of this coursework is Wednesday 21st March 2012. Any coursework handed in later than *2 working days* after the deadline will automatically receive a zero mark.
