CS 4240: Compilers and Interpreters
Phase 2: Parse tree, Symbol table, Semantics, and IR
Due Date: March 24, 2014

This phase of the project consists of four parts. You will be adding abstract syntax tree generation during the parsing step, creating a symbol table, doing several semantic checks, and generating intermediate code for use in the final phase.

Part 1: Parse Tree

The first step of this project is enable your parser to create a parse tree from an input program. You are to integrate this code into your current parser. We expect you to use your own code from phase 1, but if this is a problem, you can contact the TA. This part is essentially a debugging step to ensure you have the correct parse before continuing. Note that we do not expect you to create an abstract syntax tree here (which is a more succinct form that elicits unneeded nodes that were needed for parsing but which are no longer necessary). Also, to remove left recursion, you had to break the left associativity of your grammar in phase 1 of the project. In order to construct the correct parse tree, you will have to reinforce left associativity. Refer to the end of the lecture 5 slides for details.

On a correct parse, you should print out the parse tree, but you are free to decide the exact structure and details (as long as it is correct).

Part 2: Symbol Table

The symbol table is a useful structure generated by the compiler for use in other steps. Semantic checking (part 3) will make extensive use of it. The symbol table holds information about variables, constants, etc. that make up a program. Palsberg suggests the following entries for a symbol table:

- variable names
- defined constants
- defined types
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries

You must also consider the kind of information stored for each entry. Palsberg's suggestions include these:

- textual name
- data type
- dimension information (for aggregates)
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

You have some flexibility in deciding these implementation details, and the above lists are by no means requirements. The guiding principle you should use for building your symbol table and the information you keep in it should be that of utility: Store what you will need for the next parts. Your implementation will likely make use of one or more hashtables, but you are not limited to this, either.

Scoping is a key aspect of symbol tables. By this we mean two things:

1. The most closely nested rule (i.e. references always apply to the most closely nested block)
2. Declaration before use

Thus, insertion into your symbol table cannot overwrite previous declarations but instead must mask them. Subsequent lookup operations that you run against the symbol (as you do type checking in part 3, for example) should return the most recently inserted variable, definition, etc. This handles rule 1. For rule 2, a lookup operation should should never fail. If it does, it means the symbol table has not yet seen a declaration for the reference you are attempting to check.

As you consider scoping, you are free to implement your symbol table as a group of symbol tables or as one single table. For the purposes of our project, you should consider the widest scope as anything within `let-in-end`. The only nested scopes of Tiger are surrounded by `function-begin-end`. In most languages you are familiar with, declarations are permissible within functions. Tiger, however, only permits statement sequences within functions. Thus, the only variables local to functions will be those passed in the parameters. In other words, function parameters are passed by value -- with the exception of arrays, which are passed by reference -- and your scoping technique should handle this.

On a correct parse, you should print out your symbol table, but you are free to decide the exact structure and details (as long as it is correct).

Part 3: Semantic Checking

This phase consists of semantic checks. It leverages both the parse tree and symbol table built in the previous two parts. Most of your attention in this part will be focused on type checking. There are several cases in Tiger where type checking must occur:

- Agreement between binary operands
- Agreement between function return values and the function's return type
- Agreement between function calls and the function's parameters

Refer to the Tiger reference manual in phase 1 for many rules about the type semantics: the "Operators" section discusses types with respect to binary operators; the "Control Flow" section states that the `if`, `while`, and `for` expression headers should all evaluate to `int`; the "Types" section dictates that you enforce a "name type equivalence" for your types; etc. The other semantic check that must be made is with respect to return statements: A function with no return type cannot have a return statement in it.

For correct programs, your compiler should pass this part silently and emit nothing. For programs with semantic problems, your compiler should emit an error message stating the problem and relative place in the source. This may require you to add details to your parse tree indicating source location.

Part 4: Intermediate Code

The final part of this phase is to convert the program into intermediate code. For the purposes of this project, we will be using 4-address code (or "quadruple" 3-address), which has the following form:

```
op, y, z, x
```

This reads as, "Do operation op to the values y and z, and assign the new value to x." A simple example of this can be given with the following:

```
2 * a + (b - 3)
```

The IR code for this expression would be the following:

```
sub, b, 3, t1
mult, a, 2, t2
add, t1, t2, t3
```

As we can see, an important characteristic of this representation is the introduction of temporary variables, which the compiler (and therefore you, as the compiler writer) must generate. Notice that this also implies that temporary variables must now be made part of the symbol table, and their type should be derived from the result of the operation. For example, in the above expression, t1, t2, and t3 would all marked as type `int`. We say that the types for temps "propagate" through the program. You do not yet have to worry about the number of temporary variables you are creating. That is, you may assume infinitely many registers.

One subtlety with the intermediate code is handling arrays. Arrays may be multidimensional, so you will have to linearize accesses. Consider the following examples for how we expect you to calculate the offsets:

- A one dimensional array of size 5, accessed by “arr[1]” → offset is 1
- A two dimensional array of size 5x5, accessed by “arr[1][1]” → offset is 6
- A three dimensional array of size 5x5x5, accessed by “arr[1][1][1]” → offset is 31

Lastly, your intermediate code will have one exception to the 4-address structure. For instructions for function calls, you will generate an instruction very similar to that in the source. Function calls with no return values will look like the following:

```
call, func_name, param1, param2, ..., paramn
```

And function calls with return values will have a similar structure:

```
callr, x, func_name, param1, param2, ..., paramn
```

For each correct program fed into your compiler, you should emit a complete instruction stream of intermediate code for that program. You do not need to print any intermediate code for programs with errors.

To summarize, here is an instruction reference for your intermediate code:

Assignment: (op, x, y,_)

Op	Example source	Example IR
assign	a := b	assign, a, b,

Binary operation: (op, y, z, x)

Op	Example source	Example IR
add	a + b	add, a, b, t1
sub	a - b	sub, a, b, t1
mult	a * b	mult, a, b, t1
div	a / b	div, a, b, t1
and	a & b	and, a, b, t1
or	a | b	or, a, b, t1

Goto: (op, label, _, _)

Op	Example source	Example IR
goto	break;	goto, after_loop, ,

Branch: (op, y, z, label)

Op	Example source	Example IR
breq	if(a <> b) then	breq, a, b, after_if_part
brneq	if(a = b) then	brneq, a, b, after_if_part
brlt	if(a >= b) then	brlt, a, b, after_if_part
brgt	if(a <= b) then	brgt, a, b, after_if_part
brgeq	if(a < b) then	brgeq, a, b, after_if_part
brleq	if(a > b) then	brleq, a, b, after_if_part

Return: (op, x, _, _)

Op	Example source	Example IR
return	return a;	return, a, ,

Function call (no return value): (op, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
call	foo(x);	call, foo, x

Function call (with return value): (op, x, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
callr	a := foo(x, y, z);	callr, a, foo, x, y, z

Store into array: (op, array_name, offset, x)

Op	Example source	Example IR
array_store	arr[0] := a	array_store, arr, 0, a

Load from array: (op, x, array_name, offset)

Op	Example source	Example IR
array_load	a := arr[0]	array_load, a, arr, 0

Part 5: Turn-in

Recommended

You are free to implement these four parts in four separate steps or whatever way you like. Our suggestion is to construct the parse tree and symbol table together during the parsing step (i.e. the same parsing step that you did for phase 1). After a successful parse, you can then walk the parse tree and do semantic checking and intermediate code generation together.

Grading

Deliverables for phase 2:

1. Parse tree code (25 points)
2. Symbol table code (25 points)
3. Semantic checking code (20 points)
4. IR generation code (25 points)
5. Report (design internals, how to build, run, etc.) (5 points)