Course Project **4190.409 Compilers** **Spring 2016**

Phase 3: Semantic Analysis

In this third phase of the project, your job is to perform semantic analysis.

After completing the previous phase your parser prints the parsed input as an abstract syntax tree. If you have already used the provided AST skeleton (in ast.cpp/h), then you will not have to change much in the third phase. If you have used your own AST, you are encouraged to switch to the provided AST skeleton now.

The provided AST can be printed in textual and graphical form (see CAstScope::print()/ CAstScope::toDot(), and test_parser.cpp for an example how to use these methods).

Complement the code for constructing the AST and the semantical checks directly into your compiler from phase 2.

1. Completing the AST
Study the file ast.h and its implementation in ast.cpp. As usual, you can use the command

```
snuplc $ make doc
```

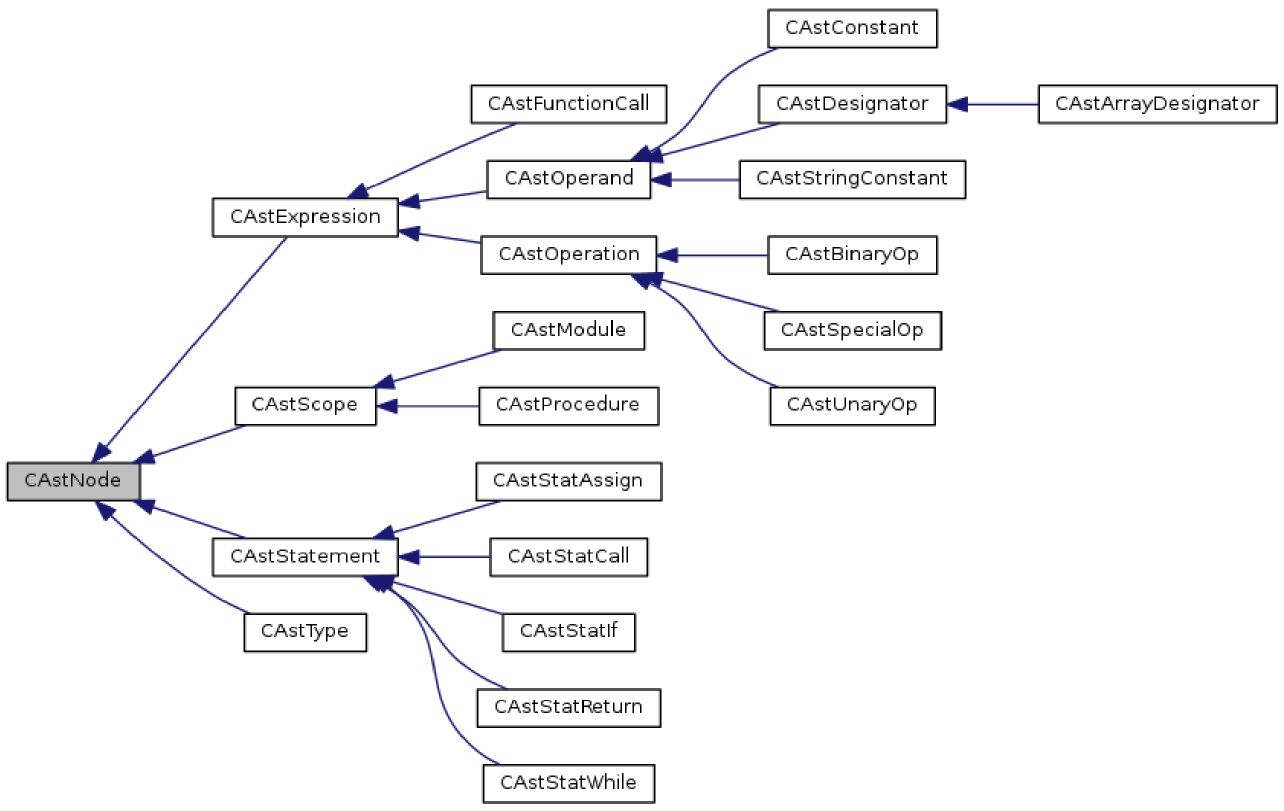
to build the Doxygen documentation for SnuPL/-1 (and the AST).

Drawing 1 shows the class diagram for the AST. During the top-down parse in the parser you can directly build the AST. Each of the parser's methods that implement a non-terminal return the appropriate AST node. Many AST nodes have a one-to-one correspondence to the parse methods of your top-down parser: the method module() of the parser returns a CAstModule class instance; a function/procedure a CAstProcedure instance, and so on.

Statement sequences are realized by using the SetNext()/GetNext() methods of CAstStatement; i.e., there is no explicit node implementing a statement sequence. This makes the class hierarchy a bit simpler, but has the disadvantage that statement lists need to be traversed explicitly wherever they can occur (i.e., in module, procedure/function, while and if-else bodies).

2. Semantic Analysis
The second part of this assignment is to perform semantic analysis. In particular, your compiler must check

- the types of the operands in expressions
- the types of the LHS and RHS in assignments
- the types of procedure/function arguments
- the number of procedure/function arguments
- catch invalid constants



Drawing 1: AST Class Diagram

The type system is described in the project outline (eTL → Compilers → Term Project → Course Project Overview). Different SnuPL/1 types are not compatible with each other at the source code level. Use the provided type manager in snuplc/src/type.[h/cpp] to obtain a reference to one of the basic types in SnuPL/1: integers, characters, booleans, and the NULL type. The respective methods are CTypeManager::Get()→GetInt()/GetChar()/GetBool()/GetNull() to get a reference to an integer, boolean or NULL (void) type.
Composite types such as arrays and pointers are also managed by the type manager. Have a look at CTypeManager::GetPointer()/GetArray().

The AST provides two methods to perform type checking:

- const CType* GetType()
- bool TypeCheck(CToken *t, string *msg)
-

CAstNode::GetType() (and implementations in subclasses) must return the type of the node. Return the SnuPL/1 NULL type (CTypeManager::Get()→GetNull()) for nodes that have no type and NULL for invalid types. Many of the GetType() methods are implemented, but you need to implement GetType() for CAstBinaryOp, CAstUnaryOp, CAstSpecialOp, CAstArrayDesignator, and CAstStringConstant.
CAstNode::TypeCheck() performs type checking. For operations, for example, type checking involves checking whether the operation is defined for the types of the left- and right-hand side of the expression and whether the two types match. For nodes representing a sequence of statements, you need to explicitly call TypeCheck() on all statements. Subroutine calls need to check actual parameters (number, types), return statements the type of the function and the provided type, and so on.

We have seen in class that type checks are performed bottom up. When implementing semantic checks, first enable the call to the type checker in CParser::Parse(). You need to implement the GetType()/TypeCheck() methods such that they

- return the correct type for the operation
- perform type checks on all statements, if the node contains a statement list
- recursively perform type checks on expressions

As an example, the code below shows the reference implementation of the type checking code for CAstScope:

```
bool CAstScope::TypeCheck(CToken *t, string *msg) const
{
  bool result = true;

  try {
    CAstStatement *s = _statseq;
    while (result && (s != NULL)) {
      result = s->TypeCheck(t, msg);
      s = s->GetNext();
    }

    vector<CAstScope*>::const_iterator it = _children.begin();
    while (result && (it != _children.end())) {
      result = (*it)->TypeCheck(t, msg);
      it++;
    }
  } catch (...) {
    result = false;
  }

  return result;
}
```

CAstModule and CAstProcedure are subclasses of CAstScope, this implementation takes care of both. The first while loop traverses the statement list and runs the type check code on each statement. The second loop traverses eventual children of the scope (in our case, only the module node will have zero or more sub-scopes in the form of procedures/functions).

Illustration 1 shows the implementation of the type checking code for return statements. The first line retrieves the type of the enclosing scope. For the module body and procedures, this should return the NULL type; only for functions the returned type shall be integer or boolean. Depending on whether a type is expected (scope type not NULL) or not, different type checks are performed.

The test program for this third phase is identical to the second phase. First enable the type checking code in CParser::Parse() first (currently commented out), then run:

```
snuplc $ make test_parser
snuplc $ ./test_parser ../test/semanal/semantics.mod
```

In the directory test/semanal/ you will find a few test files to test your semantic analysis code. We advise you to create your own test cases to test corner cases; we use our own set of test files to test (and grade) your submission.

```
bool CAstStatReturn::TypeCheck(CToken *t, string *msg) const
{
  const CType *st = GetScope()->GetType();
  CAstExpression *e = GetExpression();

  if (st->Match(CTypeManager::Get()->GetNull())) {
    if (e != NULL) {
      if (t != NULL) *t = e->GetToken();
      if (msg != NULL) *msg = "superfluous expression after return.";
      return false;
    }
  } else {
    if (e == NULL) {
      if (t != NULL) *t = GetToken();
      if (msg != NULL) *msg = "expression expected after return.";
      return false;
    }

    if (!e->TypeCheck(t, msg)) return false;

    if (!st->Match(e->GetType())) {
      if (t != NULL) *t = e->GetToken();
      if (msg != NULL) *msg = "return type mismatch.";
      return false;
    }
  }

  return true;
}
```

Illustration 1: Type checking code for CAstStatReturn

Notes

- ***Implementation location of semantical checks (parser.cpp vs. ast.cpp)***
  Certain tests are easier to implement in the parser such as the function identifier check. There is no strict rule what should go where, you are free to choose. Nevertheless, try to keep semantic analysis in the AST and perform semantic actions in the parser only if necessary.
  The reference implementation performs the following semantical check in the parser:
  - module/subroutine identifier match (in CParser::module/subroutineDecl)
  - duplicate subroutine/variable declaration (in CParser::subroutineDecl/varDeclSequence)
  - (scalar) return type for functions (in CParser::subroutineDecl)
  - declaration before use (because CAstDesignator requires a symbol; in CParser::qualident)
  - subroutine calls require a valid symbol (in CParser::subroutineCall)
  - array dimensions provided for array declarations (in CParser::type/typep)
  - array dimension constants > 0 (in CParser::type/typep)
  - integer range checks 0 ~ 2^{32}-1 (MAX_INT/MIN_INT are tested later)

Notes (cont'd)

- ***Implicit type conversions***
  There are two location where implicit type conversions have to be performed:
  - formal array parameters of type *array*<...> need to be converted to *ptr to array*<...>.
  - array arguments that are themselves not pointers need to be converted to a *ptr to array* as well.

  The reference implementation performs the former conversion in CParser::type, the latter one in CAstFunctionCall::AddArg(arg). We create a CAstSpecialOp node with the opAddress operator and one child, the 'arg' expression.
- ***Array assignments***
  SnuPL/1 does not specify whether assignments of compound types (e.g., array := array) are allowed or not. The reference implementation does not support compound types in assignments and return statements. You may choose to support them, but note that this will later require modifications to the code generator as well.
- ***Strings***
  Strings are immutable constants, but internally represented as character arrays. Strings are special in the sense that they represent *initialized* arrays. In addition, since arrays are passed by reference in SnuPL/1, you cannot pass string constants directly in function calls.
  We therefore need to generate an initialized global variable for each string constant and replace the use with a reference to that variable. The AST provides the necessary classes and methods to deal with such cases. In CAstStringConstant, create a new array of char for the string constant and associate a CDataInitString data initializer with it. Then create a new (unique) global symbol for the string and add it to the global symbol table.

The file reference/3_test_parser is a binary of our reference implementation for the third phase. You can use it to compare your parser against the reference implementation. The textual output does not need to be identical, but you should catch the same semantical errors. As usual, note that the reference implementation may still contain bugs - if you discover a bug, please let us know.

Materials to submit:

- source code of your compiler (use Doxygen-style comments)
- report describing your implementation of the type checking in the AST (PDF)

Submission:

- the deadline for the third phase is **Wednesday, May 11, 2016 before midnight**.
- email your submission to the TA (compiler-ta@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

As usual: start early, ask often! We are here to help.

Happy coding!