Phase 3: Semantic Analysis

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

In the second phase, your parser has already printed 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).

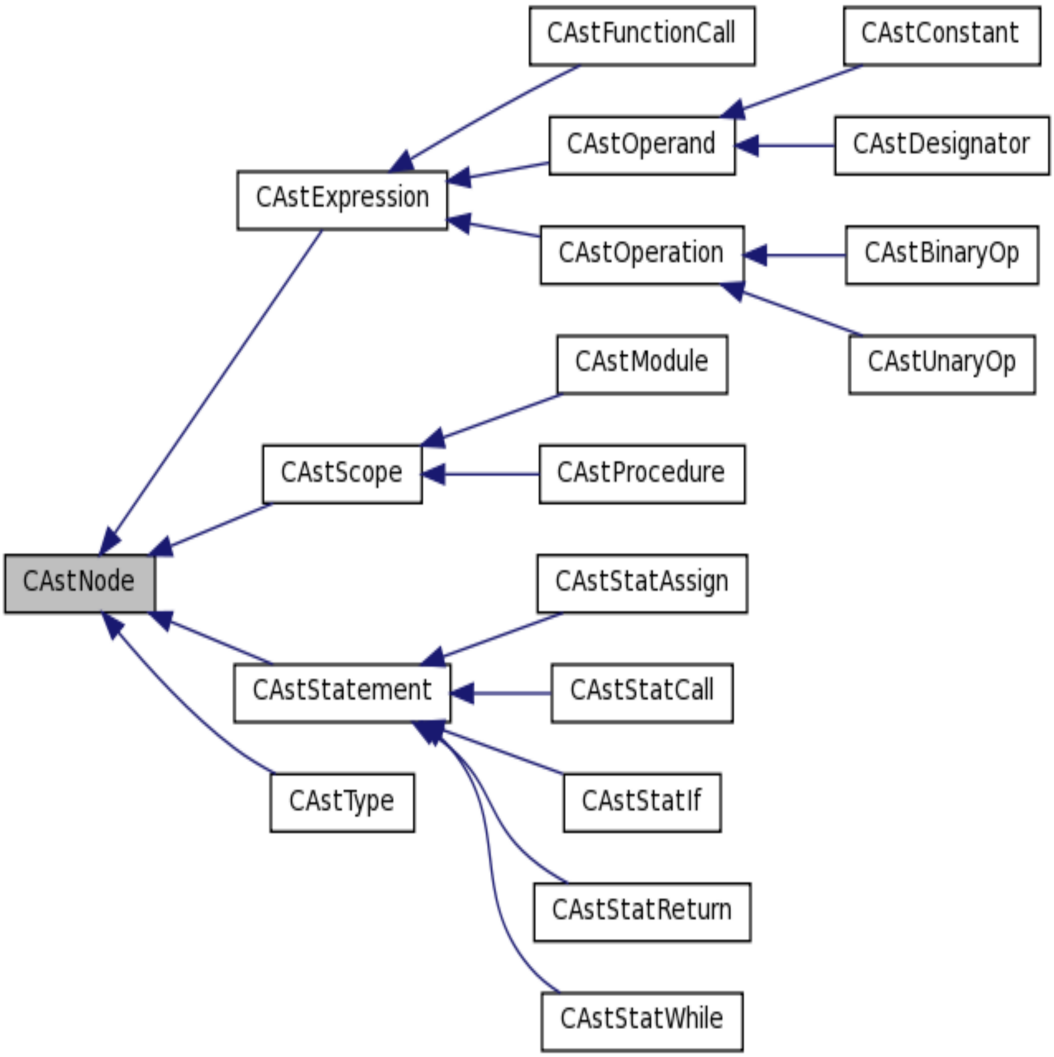
Drawing 1 shows the class diagram for the AST. During the top-down parse in the parser you can directly build the AST. Each of the parser's methods that implement a non-terminal return the appropriate AST node. Many AST nodes have a one-to-one correspondence to the parse methods of your top-down parser: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

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 listed at the end of this assignment. In particular, integers and booleans are not compatible with each other. Use the provided type manager (snuplc/src/type.[h/cpp]) to obtain a reference to the three basic types in SnuPL/0: integers, booleans, and the NULL type. The respective methods are CTypeManager::Get()->GetInt()/GetBoolean()/GetNull() to get a reference to an integer, boolean or NULL (void) type.

The AST provides two methods to perform type checking:

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

CAstNode::GetType() returns the type of each node, or the NULL type if the node does not have any type. We have already implemented GetType() in CAstNode and all subclasses, so you will only have to deal with the method TypeCheck(). We have already put all TypeCheck() methods in place, but their implementation is empty, i.e., they all return true.

We have seen in class that type checks are performed bottom up. The parser (parser.cpp) already invokes CAstModule::TypeCheck() after parsing the input. You need to implement the TypeCheck() methods such that they

- 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 subscopes 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 non NULL) or not, different type checks are performed.

The test program for this third phase is idendical to the second phase:

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

In the directory test/semanal/ you can find a test file to test your semantic analysis code. We advise you to create your own test cases to test special cases; we have our own set of test files to test (and grade) your submission.

The file snuplc/reference.parser is a binary of our reference implementation for the third phase. You may use it to compare your parser against the reference implementation. Note that the textual output does not need to be identical, but you should catch the same semantical errors.

```
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

Submission:

- the deadline for the third phase is **November 18, 2014 before midnight**.
- submit a tarball of your SnuPL/0 compiler by email 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!

Appendix: SnuPL/0 Type System

SnuPL/0 supports two scalar types, integers and booleans. The types are not compatible, and there is no type casting.

The storage size, the alignment requirements and the value range are given in the table below:

Type	Storage Size	Alignment	Value Range
integer	4 bytes	4 bytes	-2^{31} .. $2^{31}-1$
boolean	1 byte	1 byte	true, false

The semantics of the different operations for the two types are as follows:

Operator	integer	boolean
+	binary: <integer> ← <integer> + <integer> unary: <integer> ← <integer>	*n/a*
–	binary: <integer> ← <integer> - <integer> unary: <integer> ← -<integer>	*n/a*
*	<integer> ← <integer> * <integer>	*n/a*
/	<integer> ← <integer> / <integer> rounded towards zero	*n/a*
&&	*n/a*	<boolean> ← <boolean> ∧ <boolean>
||	*n/a*	<boolean> ← <boolean> ∨ <boolean>
!	*n/a*	<boolean> ← ¬ <boolean>
=	<boolean> ← <integer> = <integer>	<boolean> ← <boolean> = <boolean>
#	<boolean> ← <integer> # <integer>	<boolean> ← <boolean> # <boolean>
<	<boolean> ← <integer> < <integer>	*n/a*
<=	<boolean> ← <integer> <= <integer>	*n/a*
=>	<boolean> ← <integer> => <integer>	*n/a*
>	<boolean> ← <integer> > <integer>	*n/a*