Phase 3: Semantic Analysis

In this third phase of our term project, we implement semantic analysis and evaluated constant expressions.

After completing the previous phase your parser prints the parsed input as an abstract syntax tree. If you have used the provided AST skeleton (in ast.cpp/h), then the coding required in this third phase is moderate. If you have used your own AST, implementing all the necessary functions for type checking will cause some work. Alternatively, you can also switch to the provided AST skeleton at this point in time.

The only handout for this phase is a new Makefile and a test driver that invokes semantic analysis. The code for the semantic checks should be added directly into your compiler obtained in phase 2.

Prerequisite: 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).

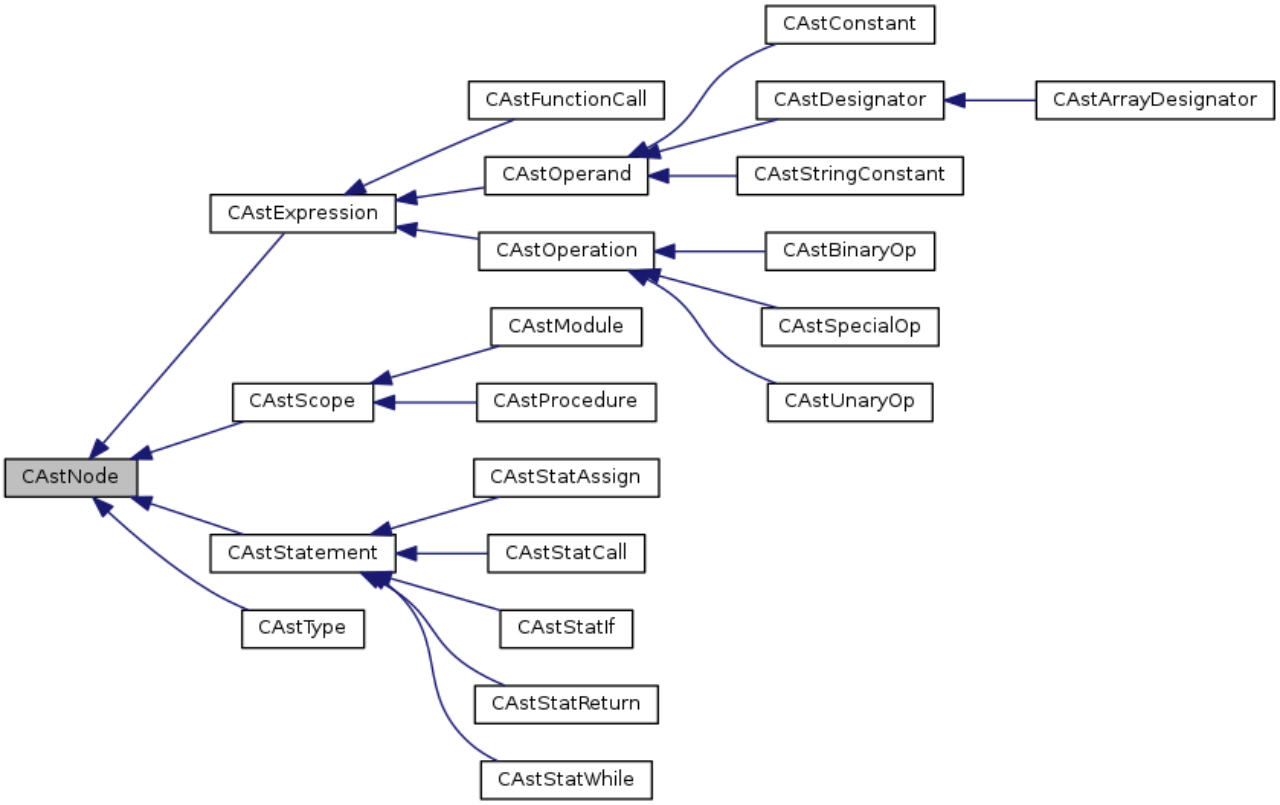
Drawing 1 shows the class diagram for the AST. During the top-down parse in the parser you can directly build the AST. Each of the parser's methods that implement a non-terminal return the appropriate AST node. Many AST nodes have a one-to-one correspondence to the parse methods of your top-down parser: 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).

1. Semantic Analysis

The first 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 type of the bound variable and its definition in the symbol table
- 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 → Project Description). Different SnuPL/2 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/2: long integers, integers, characters, booleans, and the NULL type. The respective methods are CTypeManager::Get()→GetLongint()/GetInteger()/GetChar()/GetBool()/GetNull() to obtain a reference to a long integer, integer, character, 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) perform *type inference*, i.e., these methods must return the type of the node. Scope, statement, and type nodes that have no type return the Null type (CTypeManager::Get()→GetNull()). Expression node, on the other hand, return NULL for invalid types. Many of the GetType() methods are already provided; you need to implement GetType() for CAstBinaryOp, CAstUnaryOp, CAstSpecialOp, CAstArrayDesignator, and CAstStringConstant. There are four routines to be completed in the type system (CPointerType::Match/Compare(), CArrayType::Match/Compare()).
CAstNode::TypeCheck() performs *type checking*. For binary 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.

Type checks are performed bottom up. 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 the children of the scope (in our case, only the module node can contain 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; for functions the returned type is one of the available scalar types. Depending on whether a type is expected (scope type not Null) or not, different type checks are performed.

The test driver for this third phase is built by running

snuplc $ make test_semanal
snuplc $ ./test_semanal ../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

- ***Location of semantic 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 semantic 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)
  - basic longint range checks $0 \sim 2^{64}-1$
    (type-specific range checks are performed in CAstConstant::TypeCheck())

Notes (cont'd)

- ***Implicit type conversions***
  There are two locations 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/2 does not support compound types in assignments and return statements. You may choose to relax that limitation, however, 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/2, 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.

2. Evaluation of Constants

Evaluating (and type checking) constant definitions is also performed in this third phase. Type checking is "free" after you have implemented expression evaluation; you can simply run the type checker on the constant expressions. To determine the value of the constants, the CAstExpression class and its subclasses implement the method Evalute() that computes the value of a constant at compile-time. The value of a constant is returned in a CDataInitializer; data.cpp/h defines a subclass for each supported constant data type (CDataInitInteger, CDataInitBoolean, CDataInitChar, and CDataInitString). The handout does not an initializer for longints; add it yourself.

The following code shows how the reference implementation creates the CDataInitializers for CAstConstant nodes:

```
const CDataInitializer* CAstConstant::Evaluate(void) const
{
  if (_type→IsLongint())  return new CDataInitLongint((long long)GetValue());
  if (_type→IsInteger())  return new CDataInitInteger((int)GetValue());
  if (_type→IsBoolean())  return new CDataInitBoolean((bool)GetValue());
  if (_type→IsChar())     return new CDataInitChar((char)GetValue());
}
```

In addition to constant nodes, evaluation must be implemented for other named constants (CAstDesignator), and unary and binary expressions (CAstUnary/BinaryOp). Be careful not to introduce memory leaks when dealing with CDataInitializers.

To keep things simple constant expressions are type checked and evaluated immediately after declaration in the reference compiler. The following code snippet shows the relevant part from the reference implementation:

```
void CParser::constDeclSequence(CAstScope *s)
{
  CToken t;
  CSymtab *st = s->GetSymbolTable();
  assert(st != NULL);

  do {
    // store list of constant identifiers in vlist
    vector<CToken> vlist;
    do { … };

    Consume(tColon);

    // parse type of constant and make sure it's valid
    CAstType *ctype = type(s, mConstant);
    assert(ctype != NULL);

    Consume(tRelOp, &t);
    if (t.GetValue() != "=") SetError(t, "equal operator expected.");

    // parse expression
    CAstExpression *exp = expression(s);
    assert(exp != NULL);

    Consume(tSemicolon);

    // type check: declared type vs evaluated type of expression
    const CType *etype = exp->GetType();
    if (!ctype->GetType()->Match(etype)) {
      SetError(t, "Type mismatch in constant expression.");
    }

    // evaluate expression
    const CDataInitializer *di = exp->Evaluate();
    if (di == NULL) {
      SetError(t, "Cannot evaluate constant expression.");
    }

    // add constants to symbol table
    for (size_t i=0; i<vlist.size(); i++) {
      string id = vlist[i].GetValue();
      if (st->FindSymbol(id, sLocal)) {
        SetError(vlist[i], "duplicated identifier '" + id + "'.");
      }

      st->AddSymbol(s->CreateConst(id, ctype->GetType(), di));
    }

  } while (_scanner->Peek().GetType() == tIdent);
}
```

A reference implementation can be found in snuplc/reference/test_semanal. We provide three different reference implementation to deal with the problem of negative constants; this will be explained during the lab session.

Materials to submit:

- source code of the scanner (use Doxygen-style comments)
- a report describing your implementation of the semantic analysis (PDF)
  (the report is almost as important as your code. Make sure to put sufficient effort into it!)

Submission:

- the deadline for the third phase is **Tuesday, November 3, 2020 at 14:00**.
- push your modifications to our Gitlab server and create a tag "SubmissionPhase3". The date of the last commit counts as the submission date.

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

Happy coding!