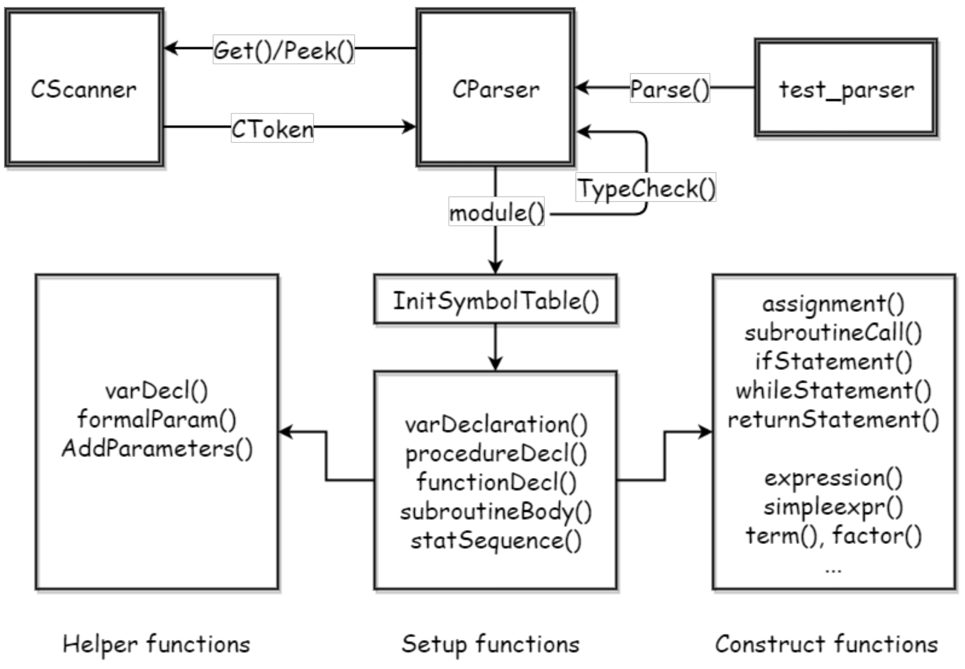
Semantic Analysis implementaion report

2013-11394 김형모
2013-11412 오평석

1 Flow diagram

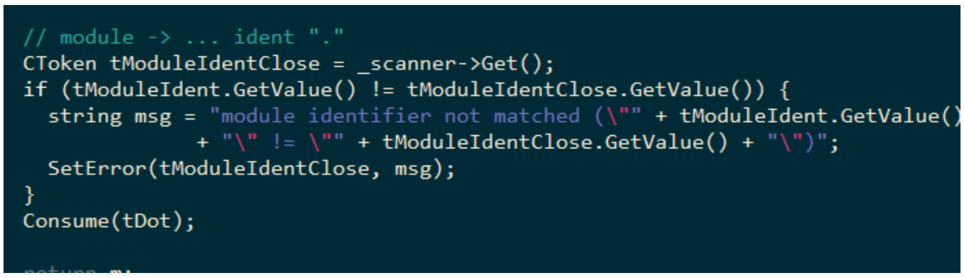
1.1 Structure

In this phase, 1) we checks syntatical issues by consuming tokens and constructs a AST(Abstract syntax tree), and 2) we checks semantic issues by checking type of nodes in constructed parse tree recursively. Some semantic issues are checked before type checking.

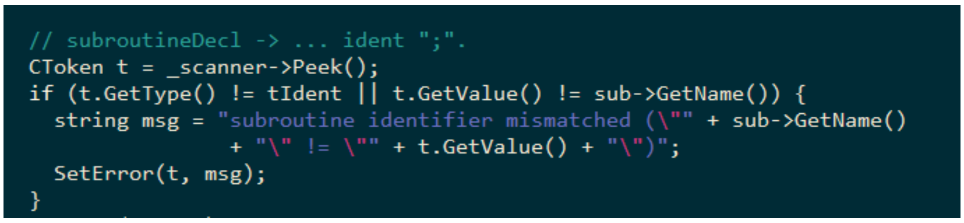


2 Improvements in parser.cpp

2.1 module



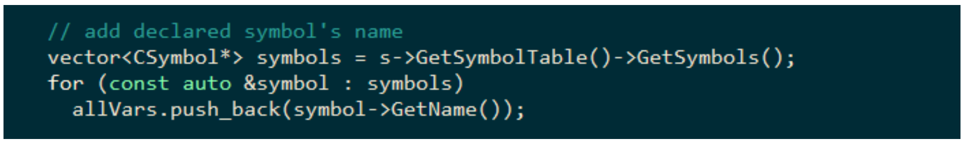
```
// module -> ... ident "."
CToken tModuleIdentClose = _scanner->Get();
if (tModuleIdent.GetValue() != tModuleIdentClose.GetValue()) {
  string msg = "module identifier not matched (\"" + tModuleIdent.GetValue(
            + "\" != \"" + tModuleIdentClose.GetValue() + "\")";
  SetError(tModuleIdentClose, msg);
}
Consume(tDot);
```



```
// subroutineDecl -> ... ident ";".
CToken t = _scanner->Peek();
if (t.GetType() != tIdent || t.GetValue() != sub->GetName()) {
  string msg = "subroutine identifier mismatched (\"" + sub->GetName()
               + "\" != \"" + t.GetValue() + "\")";
  SetError(t, msg);
}
```

We added codes that checks module/subroutine identifier match by comparing name of identifier.

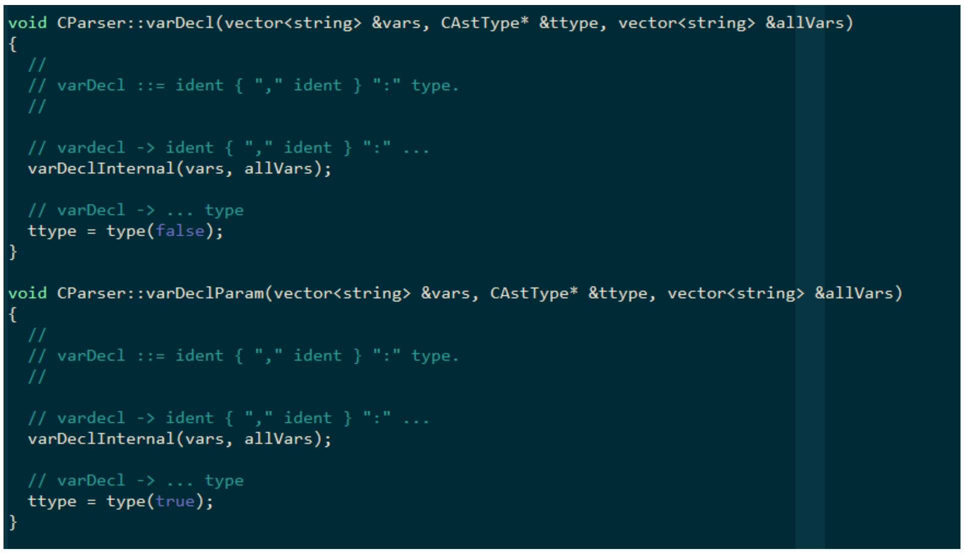
2.2 varDecl



```
  // add declared symbol's name
  vector<CSymbol*> symbols = s->GetSymbolTable()->GetSymbols();
  for (const auto &symbol : symbols)
    allVars.push_back(symbol->GetName());
```

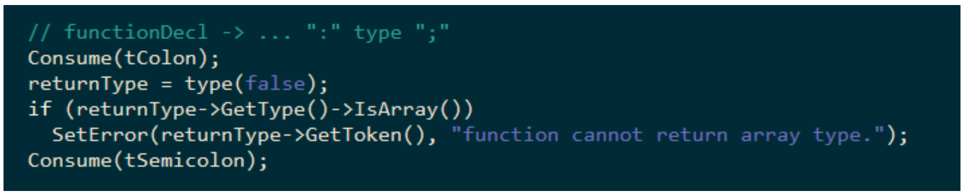
The varDecl cannot re-declare variables. In the varDeclaration, we passed all symbols in symbol table to the varDecl to ristrict variable names.

We split one varDecl into two function: varDecl and varDeclParam. The varDeclParam allows open array, but the opposit does not allow it.



```
void CParser::varDecl(vector<string> &vars, CAstType* &ttype, vector<string> &allVars)
{
  //
  // varDecl ::= ident { "," ident } ":" type.
  //

  // vardecl -> ident { "," ident } ":" ...
  varDeclInternal(vars, allVars);

  // varDecl -> ... type
  ttype = type(false);
}

void CParser::varDeclParam(vector<string> &vars, CAstType* &ttype, vector<string> &allVars)
{
  //
  // varDecl ::= ident { "," ident } ":" type.
  //

  // vardecl -> ident { "," ident } ":" ...
  varDeclInternal(vars, allVars);

  // varDecl -> ... type
  ttype = type(true);
}
```

2.3 functionDecl and formalParam



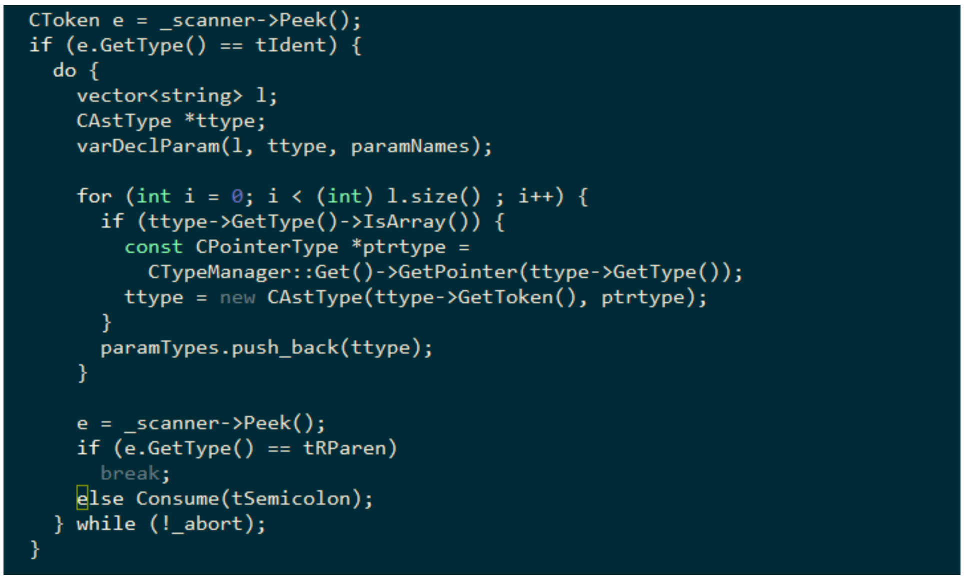
```
  // functionDecl -> ... ":" type ";"
  Consume(tColon);
  returnType = type(false);
  if (returnType->GetType()->IsArray())
    SetError(returnType->GetToken(), "function cannot return array type.");
  Consume(tSemicolon);
```

The return type for function cannot be an array. Only scala type can be returned.

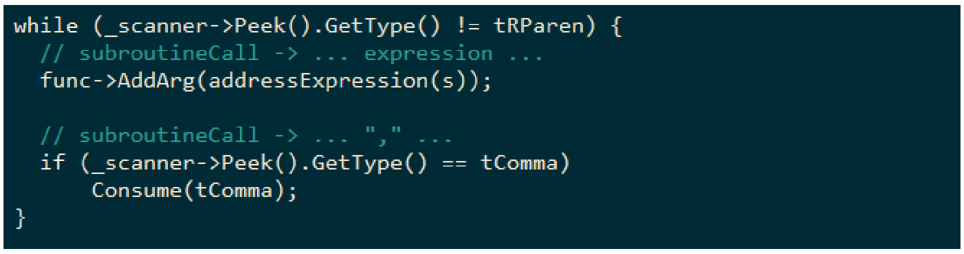
Array parameter should be casted into pointer to array type. We cast the parameter type into its poiter type when the parameter is (may be open) array.

2.4 addressing expression when subroutineCall

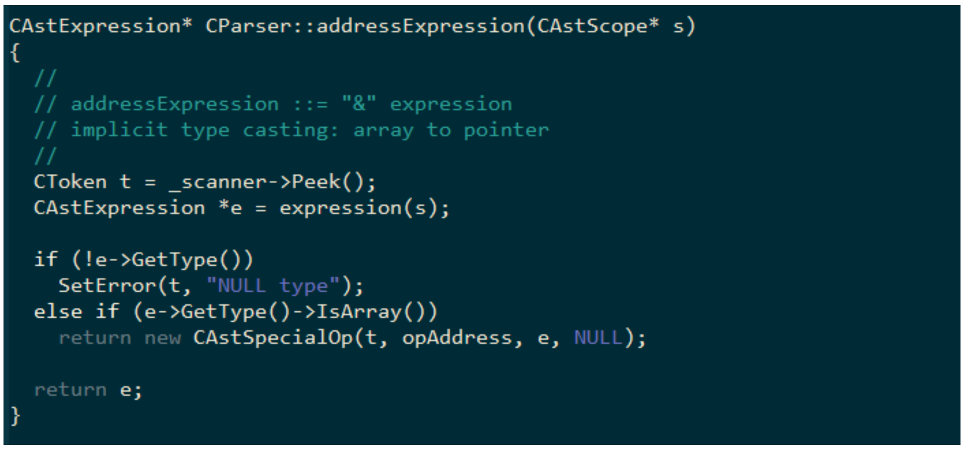
In the subroutineCall(functionCall for our implementation), arguments whose type is array should be casted into pointer to array. By using wraper function, we apply special address operation(&) selectively.



```
CToken e = _scanner->Peek();
if (e.GetType() == tIdent) {
  do {
    vector<string> l;
    CAstType *ttype;
    varDeclParam(l, ttype, paramNames);

    for (int i = 0; i < (int) l.size() ; i++) {
      if (ttype->GetType()->IsArray()) {
        const CPointerType *ptrtype =
          CTypeManager::Get()->GetPointer(ttype->GetType());
        ttype = new CAstType(ttype->GetToken(), ptrtype);
      }
      paramTypes.push_back(ttype);
    }

    e = _scanner->Peek();
    if (e.GetType() == tRParen)
      break;
    else Consume(tSemicolon);
  } while (!_abort);
}
```

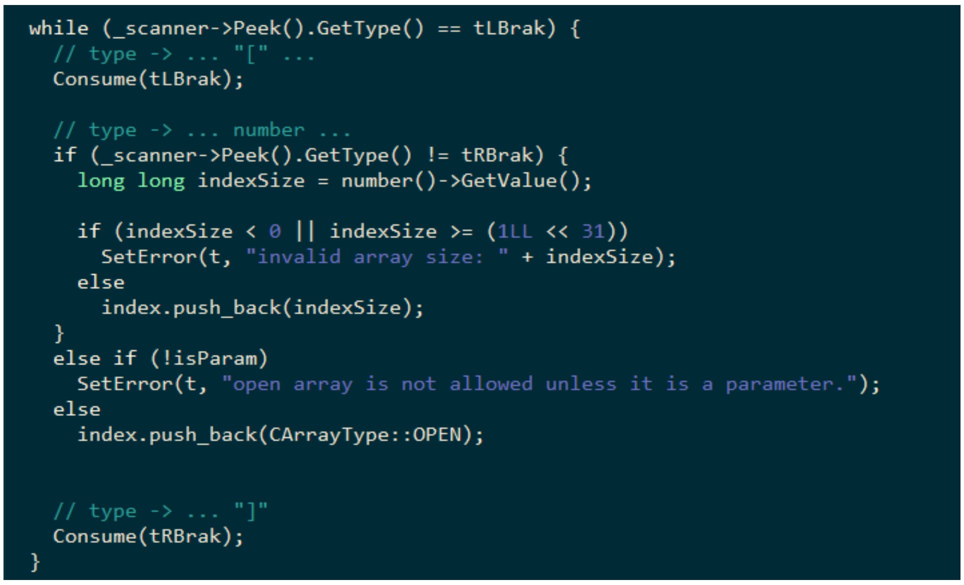


```
while (_scanner->Peek().GetType() != tRParen) {
  // subroutineCall -> ... expression ...
  func->AddArg(addressExpression(s));

  // subroutineCall -> ... "," ...
  if (_scanner->Peek().GetType() == tComma)
      Consume(tComma);
}
```



```
CAstExpression* CParser::addressExpression(CAstScope* s)
{
  //
  // addressExpression ::= "&" expression
  // implicit type casting: array to pointer
  //
  CToken t = _scanner->Peek();
  CAstExpression *e = expression(s);

  if (!e->GetType())
    SetError(t, "NULL type");
  else if (e->GetType()->IsArray())
    return new CAstSpecialOp(t, opAddress, e, NULL);

  return e;
}
```

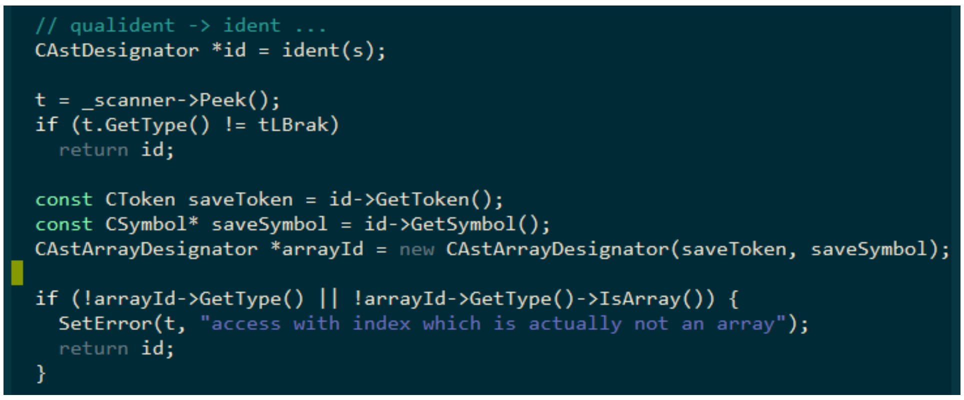
2.5 type



```
while (_scanner->Peek().GetType() == tLBrak) {
  // type -> ... "[" ...
  Consume(tLBrak);

  // type -> ... number ...
  if (_scanner->Peek().GetType() != tRBrak) {
    long long indexSize = number()->GetValue();

    if (indexSize < 0 || indexSize >= (1LL << 31))
      SetError(t, "invalid array size: " + indexSize);
    else
      index.push_back(indexSize);
  }
  else if (!isParam)
    SetError(t, "open array is not allowed unless it is a parameter.");
  else
    index.push_back(CArrayType::OPEN);


  // type -> ... "]"
  Consume(tRBrak);
}
```

When determine type of array variables, open array is allowed only when the variable is in formalParam. Using condition variable ‘isParam’, the parser checks open array exception.

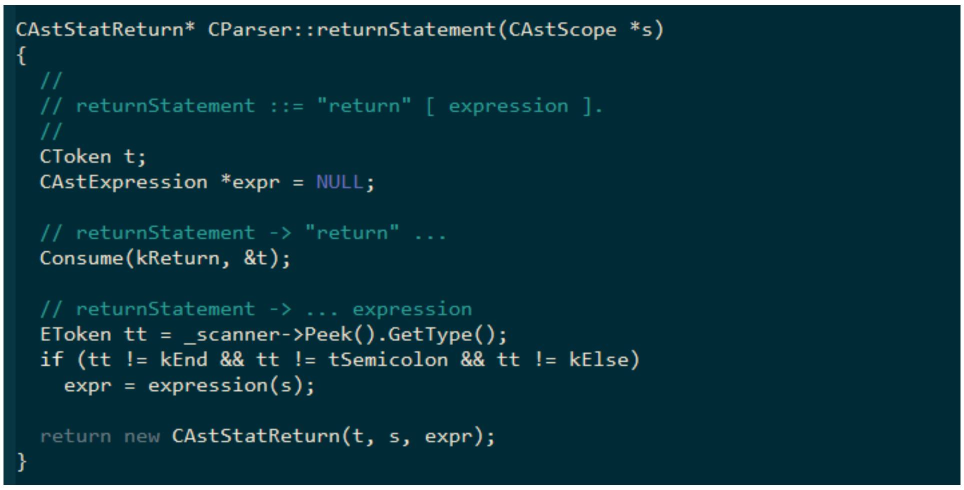
2.6 qualident



```
// qualident -> ident ...
CAstDesignator *id = ident(s);

t = _scanner->Peek();
if (t.GetType() != tLBrak)
  return id;

const CToken saveToken = id->GetToken();
const CSymbol* saveSymbol = id->GetSymbol();
CAstArrayDesignator *arrayId = new CAstArrayDesignator(saveToken, saveSymbol);

if (!arrayId->GetType() || !arrayId->GetType()->IsArray()) {
  SetError(t, "access with index which is actually not an array");
  return id;
}
```

Access with braket to identifier which is not an array, function GetType() returns NULL for invalid type error. Qualident catches it and raise parsing error.

2.7 returnStatement



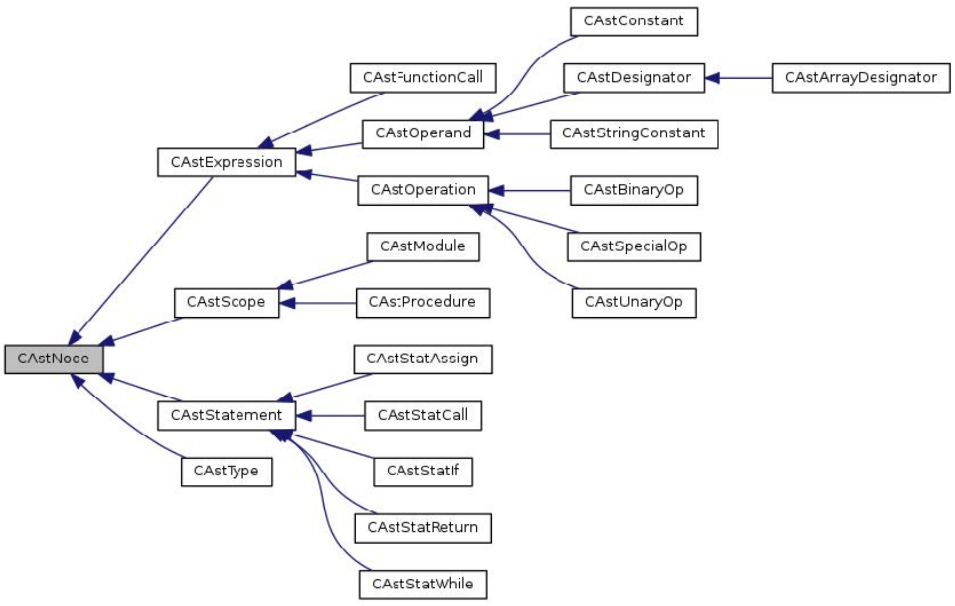
```
CAstStatReturn* CParser::returnStatement(CAstScope *s)
{
  //
  // returnStatement ::= "return" [ expression ].
  //
  CToken t;
  CAstExpression *expr = NULL;

  // returnStatement -> "return" ...
  Consume(kReturn, &t);

  // returnStatement -> ... expression
  EToken tt = _scanner->Peek().GetType();
  if (tt != kEnd && tt != tSemicolon && tt != kElse)
    expr = expression(s);

  return new CAstStatReturn(t, s, expr);
}
```

We ommited ‘else’ keyword for FOLLOW(returnStatement), so it is added. This enables to pass ‘return statement right before else keyword’.

3 Typechecking in ast.cpp

3.1 Structure

Here is the inheritance diagram for CAstNode class. This is identical to skeleton code's.



Each class can have GetType method and TypeCheck method.

bool TypeCheck (CToken *t, string *msg) : perform type checking

const CType* GetType (void) : return the type of this node

Except for trivial cases and implementations, all TypeCheck and GetType methods that we implemented in src/ast.cpp are following:

3.2 CAstScope

3.2.1 TypeCheck

1. perform its type checking for each statement consists of statSequence
2. perform its type checking for each child scope of this scope.

3.3 CAstProcedure

3.4 GetType

procedure/function's return type

3.5 CAstStatAssign

3.5.1 TypeCheck

1. perform type checking for lhs and rhs.
2. check the type of lhs is not scalar type.
3. check the type of rhs is not scalar type.
4. check the type of matches with the type of rhs.

3.5.2 GetType

return the type of lhs.

3.6 CAstStatCall

3.6.1 TypeCheck

Since CAstStatCall has corresponsive functionCall, it just performs functionCall's type checking.

3.7 CAstStatReturn

3.7.1 TypeCheck

Procedure: Checks if there is no expression right after return.

Function: Checks if there is an expression right after return.

Perform type checking for expression, and check its type matches with procedure/function's type.

If there is no return statement in function body, our parser does not report it is error. We decided to it as undefined behaviour like c and c++, since it is another problem to check that there are one more return statements in all cases. This is a area of program analysis.

3.7.2 GetType

If expression exists, return expression's type. If not, return null type.

3.8 CAstStatIf

3.8.1 TypeCheck

First, perform type checking for condition expression and check whether its type is boolean. Next, perform type checking for each statement in ifBody. Finally, perform type checking in elseBody similarly. The last one can be missing.

3.9 CAstStatWhile

3.9.1 TypeCheck

It is similar to TypeCheck in CAstStatIf. The only difference is that there is no elseBody in CAstStatWhile.

3.10 CAstBinaryOp

3.10.1 TypeCheck

1. Perform type checking for both operands
2. Check whether its type is one of char, int, boolean type for both operands
3. Check the type of left operand equals to the type of right operand.
4. Check the operation is defined for both operand's type.

3.10.2 GetType

If the operation is '+', '-', '*', or '/', it returns int type. Else if the operation is '&&', '||', '=', '#', '<', '<=', '>', or '>=', it returns boolean type. Else then it is an error case. (should not be happen)

3.11 CAstUnaryOp

3.11.1 TypeCheck

Perform type checking for its operand. Interestingly, there is some cases that ignore the operand's type checking result.

1. Type checking successes. Then check the type of the operand is int type in cases the operation is "+" or "-". If the operation was "!", check the type of the operand is boolean type.
2. Type checking fails. Then check the operand's node is CAstConstant. If so and the unary operator is "-", ignore the result of operand's type checking and report the type checking successes. It allows to handle INT_MAX + 1 problem as we mentioned in previous report. For more details, see 4.x and 4.x.

3.11.2 GetType

If the operation is "+" or "-", it returns int type. Else if the operation is "!", it returns boolean type. Else then it is an error case. (should not be happened)

3.12 CAstSpecialOp

3.12.1 TypeCheck

perform type checking for the operand. If the operation is opDeref, check the operand is pointer type.

3.13 GetType

opAddress(&): it returns pointer type to the operand's type.

opDeref(*): it returns the dereference type of the operand's type.

opCast((τ)): it returns the casting type.

3.14 CAstFunctionCall

3.14.1 TypeCheck

First, check the parameter number of the signature is identical to subroutineCalls'. Second, Perform type checking for each expression which is passed as parameter, and check its type matches with the parameter's type of the signature.

3.15 CAstDesignator

3.15.1 TypeCheck

Since its type is determined when it is constructed, its TypeCheck just check its type is null or invalid type.

3.15.2 GetType

return the corresspondence symbol's data type.

3.16 CAstArrayDesignator

3.16.1 TypeCheck

Perform type checking for each expression when be used to access array as index, and check its type is integer type.

3.16.2 GetType

Get the corresspondence symbol's data type. If its type is pointer type, casts it as array type. Then it returns approperiate type by counting the number of index.

For example, the symbol's type is "pointer to array of array of char" and we access the symbol like as a[0],

1. pointer to array of array of char → array of array of char.
2. array of array of char → array of char.
3. returns array of char.

3.17 CAstConstant

3.17.1 TypeCheck

Check its type is null type or invalid. If the value is 2^{31}, it returns false. Note that this return value can be ignored later.

3.17.2 GetType

Since constant's type is determined when we construct parsing tree, it just returns _type.

3.18 CAstStringConstant

3.18.1 TypeCheck

Check its type is null type or invalid.

3.18.2 GetType

Since string constant's type is determined when we construct parsing tree, it just returns _type.

4 Specific error cases

4.1 In parser.cpp

Checking in parse tree construction.

4.1.1 module/subroutine identifier mismatched

Catches opening and closing identifier mismatch error when consuming the closing indentifie by compare their value(string).

4.1.2 re-declaration varaible/procedure/function

There are four cases:

1. local - local
2. param - local
3. param - param
4. global - global

These are catched by varDecl(Internal) and varDeclaration. For the first, third, and last one, varDecl(Internal) carries all variable name that are declared by varDecl(Internal) itself to identify that next variable is duplicated or not. For the second one, VarDeclaration inserts symbols in symbol table into symbol name list of varDecl. This makes varDecl can catches duplication between parameter and local variable.

4.1.3 undeclared variable

Ident lookups symbol table(first in local, second in global). So when using undeclared variable, because of its absence in symbol table, Ident can catch this error by checking its existance in symbol table.

4.1.4 return array type

Function can only return scala type. In the subroutineCall, a condition statement checks retured value's type wether it is array type or not.

4.1.5 open array type

Only parameter's type can be an open array. Passing an boolean 'isParam' to type function, type allows/forbids open array, by checking right bracket right after left bracket.

4.1.6 indexing with variable whose type is not an array

For example, using v[4] with var v : integer; is an error. This feature is carried by checking type of arrayDesignator version of the variable when constructing designator node.

4.1.7 invalid number ($> 2^{31}$)

When constructing constant node, if number value is greater than 2^{31}, the parser raises error before type checking. If number value is exactly 2^{31}, checking boundary is postponed to typechecking phase.

4.2 In ast.cpp

Checking after construction of parse tree.

4.2.1 assignment type mismatch

Assignment is allowed only for char, boolean, and integer type. Futhermore, the type of lhs and rhs must be same.

4.2.2 procedure with return value

procedure cannot have return type except null type. So if there is some expression right after return keyword, our parser will report error.

4.2.3 function with return Null

Function does not return null type. So it is an error case when return expression is missing.

4.2.4 function return type mismatch

If the return type of the function does not match with the declared return type, it is an error.

4.2.5 condition type mismatch in if/while statement

The type of condition expression should be boolean.

4.2.6 binary operation type mismatch

The type of lhs and rhs must be same and must be match with operator's required type.

4.2.7 unary operation type mismatch

The type of operand must be match with operator's required type.

4.2.8 subroutine parameter number mismatch

For example, if the signature is "procedure f (a : integer, b : integer)", f(), f(3), f(3,4,5), f(6,7,8,9) are all error.

4.2.9 subroutine parameter type mismatch

For example, if the signature is "procedure f (a : integer, b : integer)", f(5, true), f('a', 4), f("hello", "world") are all error.

4.2.10 array index number type mismatch

We can access the value in array by using integer index only. So the following cases are error. For example,

```
a['c'] := 5
a[true] := 4
a[''hello''] := 3
```

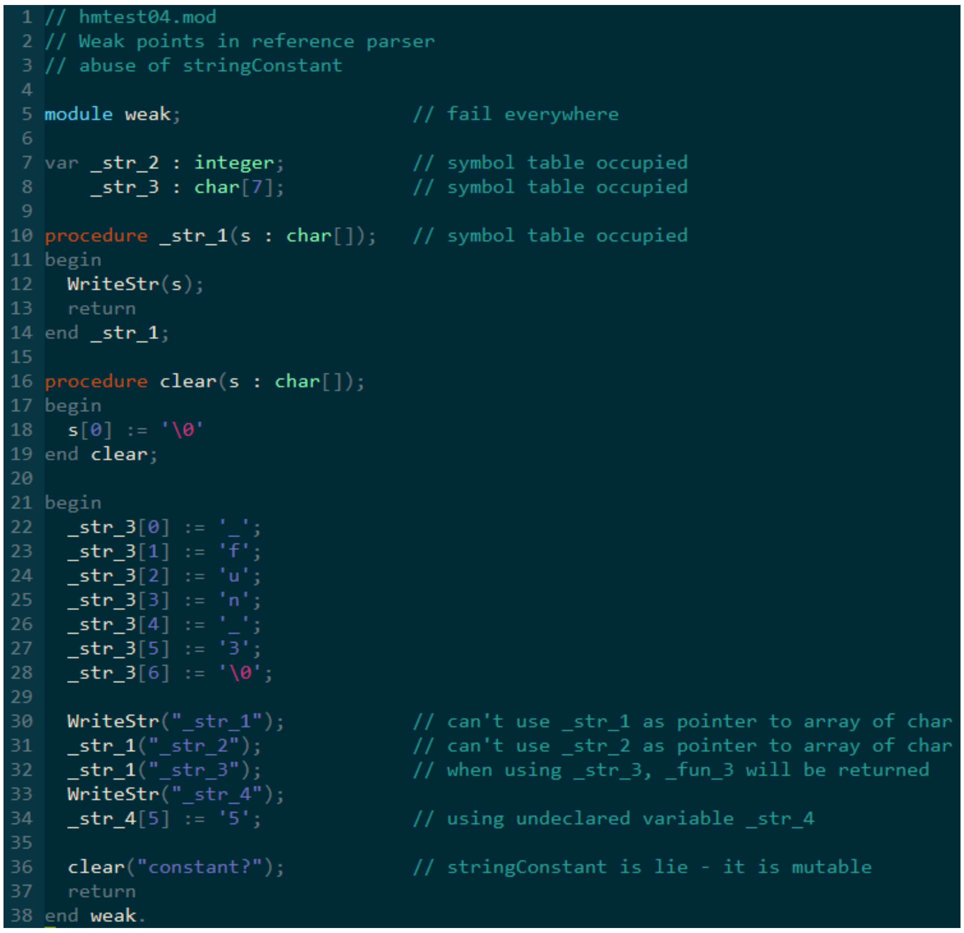
4.2.11 invalid number ($= 2^{31}$)

We adopt 'simple' policy. If the number value is exactly 2^{31}, typechecking fails. But when unary '-' operation is attatched to number node directly, typechecking passes. For example, $-2147483648 + 1$ is correct input because opNeg is boxing 2^{31}. If unary '-' is not attatched directly to the number node, for example, $-2147483648 * 1$ is error case. This outputs same with reference parser (simple version)

4.3 Error testcases

You can examine the output of the above error cases in `test/semanal/error` directory.

5 Issues with stringConstant



```
// hmtest04.mod
// Weak points in reference parser
// abuse of stringConstant

module weak;                    // fail everywhere

var _str_2 : integer;           // symbol table occupied
    _str_3 : char[7];           // symbol table occupied

procedure _str_1(s : char[]);   // symbol table occupied
begin
  WriteStr(s);
  return
end _str_1;

procedure clear(s : char[]);
begin
  s[0] := '\0'
end clear;

begin
  _str_3[0] := '_';
  _str_3[1] := 'f';
  _str_3[2] := 'u';
  _str_3[3] := 'n';
  _str_3[4] := '_';
  _str_3[5] := '3';
  _str_3[6] := '\0';

  WriteStr("_str_1");           // can't use _str_1 as pointer to array of char
  _str_1("_str_2");             // can't use _str_2 as pointer to array of char
  _str_1("_str_3");             // when using _str_3, _fun_3 will be returned
  WriteStr("_str_4");
  _str_4[5] := '5';             // using undeclared variable _str_4

  clear("constant?");           // stringConstant is lie - it is mutable
  return
end weak.
```

Above example shows some problems in implementation of `SnuPL_1`. This code passes reference parser. Abuse of these features can induce segmentation fault or some not intended behavior of functions.