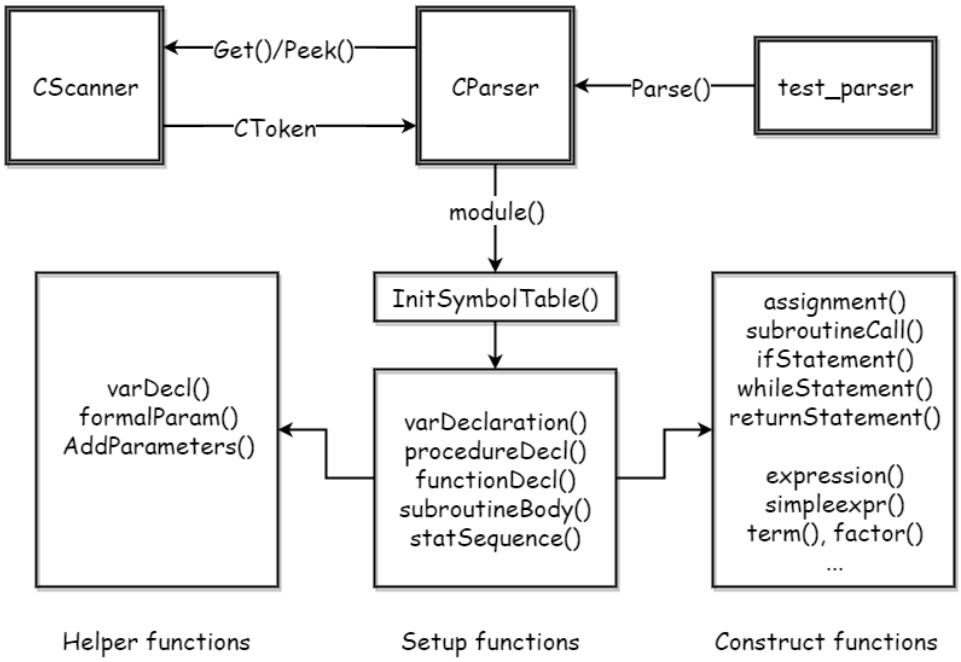
Parser implementaion report

2013-11394 김형모
2013-11412 오평석

1 Base spec

1.1 Structure

Our parser is based on skeleton(for Snupl/-1), but more complicate. In this phase, we checks syntatical issues by consuming tokens and constructs a AST(Abstract syntax tree).



1.2 Prototypes

Followings are prototypes for important methods in parser.cpp.

CAstModule *	module (void) build up AST moudle scope
void	varDeclaration (CAstScope *s) create and add symbols which are declared variables
CAstProcedure *	procedureDecl (CAstScope *s) build up AST procedure scope by procedure declaration
CAstProcedure *	functionDecl (CAstScope *s) build up AST procedure scope function declaration
void	formalParam (vector<string> &ns, vector<CAstType *> &ts) store parameter's information to add symbols to symbol table
void	subroutineBody (CAstScope *s) build up subroutineBody and set it to subroutine's statSequence
CAstStatement *	statSequence (CAstScope *s) build up AST statement node by statement sequence
CAstStatAssign *	assignment (CAstScope *s) build up AST assignment node by assignment statement
CAstStatCall *	subroutineCall (CAstScope *s) build up AST procedure call node by subroutine call
CAstFunctionCall *	functionCall (CAstScope *s) build up AST function call node by function call
CAstStatIf *	ifStatement (CAstScope *s) build up AST if-else node by if-else statement
CAstStatWhile *	whileStatement (CAstScope *s) build up AST while node by while statement
CAstStatReturn *	returnStatement (CAstScope *s) build up AST return node by return statement
CAstExpression *	expression (CAstScope *s) build up AST expression node by expression
CAstExpression *	simpleexpr (CAstScope *s) build up AST expression node by simple expression
CAstExpression *	term (CAstScope *s) build up AST expression node by term
CAstExpression *	factor (CAstScope *s) build up AST expression node by factor
CAstType *	type (void) build up AST type node by given type
CAstDesignator *	qualident (CAstScope *s) build up AST designator node by qualident
CAstDesignator *	ident (CAstScope *s) build up AST designator node by ident
CAstConstant *	number (void) build up AST constant operand node by given number
CAstConstant *	boolean (void) build up AST constant operand node by given boolean
CAstConstant *	character (void) build up AST constant operand node by given character
CAstStringConstant *	stringConst (CAstScope *s) build up AST string constant operand node by given string

2 Detail spec

2.1 FIRST/FOLLOW Set

We constructed function table for FIRST and FOLLOW. These functions allow us to build a parsing table for SnuPL/1, which is important for making

a predictive parser. We use OCaml language for generating follow entries.

X	FIRST(X)
module	`module`
type	`boolean char string`
qualident	`ident`
factor	`ident number true false char string ( !`
term	`ident number,true false char string ( !`
simpleexpr	`ident number true false char string ( ! + -`
expression	`ident number true false char string ( ! + -`
assignment	`ident`
subroutineCall	`ident`
ifStatement	`if`
whileStatement	`while`
returnStatement	`return`
statement	`ident if while return`
statSequence	`ident if while return` ϵ
varDeclaration	`var` ϵ
varDeclSequence	`ident`
varDecl	`ident`
subroutineDecl	`procedure function`
procedureDecl	`procedure`
functionDecl	`function`
formalParam	`(`
subroutineBody	`var begin`

Because the FIRST(assignment) and FIRST(subroutineCall) have `ident` together, we need to use more information. So, our parser is not pure LL(1) parser.

Also, because the FOLLOW(varDeclSequence) and FOLLOW(varDecl) have `;` together, we need to modify some productions to parse simply.

X	FOLLOW(X)
module	`EOF`
type	`) ; [`
qualident	`; , = # < <= > >= ) ] + - || * / && := else end`
factor	`; , = # < <= > >= ) ] + - || * / && else end`
term	`; , = # < <= > >= ) ] + - || else end`
simpleexpr	`; , = # < <= > >= ) ] else end`
expression	`; , ) ] else end`
assignment	`; else end`
subroutineCall	`; , = # < <= > >= ) ] + - || * / && else end`
ifStatement	`; else end`
whileStatement	`; else end`
returnStatement	`; else end`
statement	`; else end`
statSequence	`else end`
varDeclaration	`procedure function begin`
varDeclSequence	`; )`
varDecl	`; )`
subroutineDecl	`procedure function begin`
procedureDecl	`var begin`
functionDecl	`var begin`
formalParam	`: ;`
subroutineBody	`ident`

2.2 Code details

There are some hard issues to appropriately handle with phase 2. These are more difficult than just consume tokens and make AST directly.

2.2.1 create and add symbols to proper scope's symbol table

In module and subroutineBody, we call varDeclaration() to retrieve a bunch of variables at once because the type information for the variables is at the end of varDeclSequence. We create and add symbols to s' symbol table. For module, s is CAstModule's symbol table. For subroutineBody, s is CAstProcedure's symbol table.

```
void CParser::varDeclaration(CAstScope *s)
{
  //
  // varDeclaration ::= [ "var" varDeclSequence ";" ].
  // varDeclSequence ::= varDecl { ";" varDecl }.
  //

  // varDeclaration -> "var" ...
  EToken tt = _scanner->Peek().GetType();
  if (tt == kVar) {
    Consume(kVar);

    // varDeclaration -> ... varDeclSequence ...
    vector<string> allVars;
    do {
      vector<string> l;
      CAstType *ttype;

      // varDeclSequence -> ... varDecl ...
      varDecl(l, ttype, allVars);

      for (const auto &str : l) {
        CSymbol *var = s->CreateVar(str, ttype->GetType());
        s->GetSymbolTable()->AddSymbol(var);
      }

      // varDeclaration -> ... ";"
      // varDeclSequence -> ... ";" ...
      Consume(tSemicolon);
      tt = _scanner->Peek().GetType();
    } while (tt != kProc && tt != kFunc && tt != kBegin);
  }
}
```

2.2.2 construct subroutine parameters

When we construct CAstProcedure for procedure/function, we carefully handle parameter symbol. Parameter symbol needs to be added not only procedure symbol (CSymProc), but also procedure scope's symbol table.

The vector paramNames and paramTypes have the informations about all parameters. Our parser get paramNames and paramTypes from calling formalParam method.

2.2.3 distinct subroutineCall & assignment from statement

When we look the EBNF syntax definition of SnuPL/1, we found interesting things. Look at the following production rules.

```
statement ::= assignment | subroutineCall | ...
```

```
// procedureDecl -> ... ";"
Consume(tSemicolon);

CSymProc *symbol =
  new CSymProc(procedureName, CTypeManager::Get()->GetNull());
s->GetSymbolTable()->AddSymbol(symbol);

CAstProcedure *ret = new CAstProcedure(t, procedureName, s, symbol);
AddParameters(ret, symbol, paramNames, paramTypes);
```

```
// functionDecl -> ... ":" type ";"
Consume(tColon);
returnType = type();
Consume(tSemicolon);

CSymProc *symbol = new CSymProc(functionName, returnType->GetType());
s->GetSymbolTable()->AddSymbol(symbol);

CAstProcedure *ret = new CAstProcedure(t, functionName, s, symbol);
AddParameters(ret, symbol, paramNames, paramTypes);
```

```
void CParser::AddParameters
  (CAstScope *s, CSymProc *symbol, vector<string> &paramNames, vector<CAstType*> &paramTypes)
{
  int cnt = 0;

  // create parameter symbols and add them to symbol table for subroutine
  for (int i = 0; i < (int) paramNames.size() ; i++) {
    string &paramName = paramNames[i];
    CAstType* paramType = paramTypes[i];

    CSymParam* param = new CSymParam(cnt++, paramName, paramType->GetType());
    symbol->AddParam(param);
    s->GetSymbolTable()->AddSymbol(param);
  }
}
```

```
assignment ::= qualidnet | ...
qualident ::= ident { ‘‘[‘‘ expression ‘‘]’’ }.
subroutineCall ::= ident ‘‘(‘‘ [ expression {‘‘,’’ expression} ] ‘‘)’’.
```

Ident is not only the element of FIRST(assignment), but also the element of FIRST(subroutineCall). Now suppose we face tIdent token when parsing the statement. There is no way to distinguish between assignment and subroutineCall without viewing the next token.

To solve this, our parser check the symbol table when it faces that situation. If the symbol is not appeared in the symbol table, it is ‘‘`undeclared variable`’’ error. If the symbol type is procedure, the token belongs to sub-

routineCall. Otherwise, it belongs to assignment.

```
// stateSequence -> ... statement ...
tt = _scanner->Peek();
switch (tt.GetType()) {
  // statement -> assignment | subroutineCall
  case tIdent:
    {
      const CSymbol *sym = s->GetSymbolTable()->FindSymbol(tt.GetValue(), sLocal);
      if (!sym) sym = s->GetSymbolTable()->FindSymbol(tt.GetValue(), sGlobal);
      if (!sym) SetError(tt, "undeclared variable \"" + tt.GetValue() + "\"");

      ESymbolType stype = sym->GetSymbolType();
      if (stype == stProcedure) st = subroutineCall(s);
      else st = assignment(s);
    }
    break;
```

2.2.4 distinct varDeclSeqeuence & VaeDecl with semicolon

Similarly, we can not distinguish the semicolon in varDeclaration and varDeclSequence.

```
varDeclaration = [ ‘‘var’’ varDeclSequence’’ ‘‘;’’ ].
varDeclSequence = varDecl { ‘‘;’’ varDecl }.
```

Suppose we parse varDecl and face semicolon. Do we prepare to accept next varDecl, or to close varDeclaration? We don’t distinguish two cases without checking the next token. So we modified EBNF syntax definition of SnuPL/1 by replacing varDeclSequence and adapted it to our parser. Be careful about varDeclSequence is appeared not only in varDeclaration, but also in formalParam.

```
varDeclaration -> [ ‘‘var’’ varDecl { ‘‘;’’ varDecl } ‘‘;’’ ].
formalParam = ‘‘(‘‘ [ varDecl { ‘‘;’’ varDecl } ] ‘‘)’’.
```

2.2.5 initialize pre-defined functions

In SnuPL/1, there are several pre-defined functions. We added these functions to global symbol table in CParser::InitSymbolTable method.

```
void CParser::InitSymbolTable(CSymtab *s)
{
  CTypeManager *tm = CTypeManager::Get();
  CSymProc *fun;

  // function DIM(array: pointer to array; dim: integer): integer
  fun = new CSymProc("DIM", tm->GetInt());
  fun->AddParam(new CSymParam(0, "arr", tm->GetPointer(tm->GetNull())));
  fun->AddParam(new CSymParam(1, "dim", tm->GetInt()));
  s->AddSymbol(fun);

  // function DOFS(array: pointer to array): integer;
  fun = new CSymProc("DOFS", tm->GetInt());
  fun->AddParam(new CSymParam(0, "arr", tm->GetPointer(tm->GetNull())));
  s->AddSymbol(fun);
```

3 Policy spec

3.1 Error cases

3.1.1 module identifier mismatched

Our parser throws error when module identfier is mismatched. When the parser build up AST module node, it checks the given two identifier are equal. If it is mismatched, our parser will print ‘‘`module identifier not matched`’’.

```
1 // error01
2
3 module error01;
4
5 begin
6
7 end __error01__.
```

```
parsing '../test/parser/error01.mod'...
parse error : at 7:5 : module identifier not matched


Done.
```

3.1.2 procedure/function identifier mismatched

Similarly our parser reports error when function/procedure identifier is mismatched, which prints ‘‘`subroutine identifier mismatched`’’.

```
1 // procedure identifier mismatched
2
3 module p;
4
5 procedure input();
6 begin
7     ReadInt()
8 end intput;
9
10 begin
11     input()
12 end p.
```

```
parsing '../test/parser/error10.mod'...
parse error : at 8:5 : subroutine identifier not matched


Done.
```

3.1.3 undeclared variable

Our parser forbid the usage of undeclared variable. When the code try to access variable in body, our parser check that its symbol exist in local/global symbol table. If not, our parser report the error ‘‘`undeclared variable ‘name’`’’.

3.1.4 re-declaration variable/procedure/function

Our parser do not permit re-declaration variable. When our parser face duplicated definition of variable, procedure or function, the parser throws ‘‘`re-declaration variable/procedure/function ‘name’`’’.

```
1 // undeclared variable
2
3 module topdown;
4
5 var a,b : integer;
6
7 begin
8     WriteInt(a+b+c)
9 end topdown.
```

```
parsing '../test/parser/error09.mod'...
parse error : at 8:18 : undeclared variable "c"


Done.
```

```
 1 // re-definition error
 2
 3 module pstest05;
 4
 5 var a,b,c,d,a : integer;
 6
 7 begin
 8     WriteStr("Hello, World\n")
 9 end pstest05.
10
```

```
parsing '../test/parser/error02.mod'...
parse error : at 5:13 : re-declaration variable "a"


Done.
```

3.2 Unhandled cases

3.2.1 pointer to array implicit type casting

When array is passed to function/procedure, they are implicitly passed as references. Our parser has the method to do this, but we realize that this can be done smartly when we perform type checking. So we delayed it for phase 3. This is also applied when our parser face string token.

```
// re-definition error

module pstest07;

var a,b,c : integer;
    d,e,f : integer;

procedure f;
begin
end f;

begin
    WriteStr("Goodbye, World!\n");
end pstest07.
```

```
parsing '../test/parser/error04.mod'...
parse error : at 8:11 : re-declaration procedure "f"


Done.
```

```
// re-declaration function

module super_fantasy_war;

var f : integer;

function f(a,b : integer) : integer;
begin
    return a+b
end f;

begin
    WriteInt(f(1,2))
end super_fantasy_war.
```

```
parsing '../test/parser/error08.mod'...
parse error : at 7:10 : re-declaration function "f"


Done.
```

3.2.2 INT_MAX + 1

To catch this error, we should examine it has the unary operator. Also, this is error only when the unary operator is + or it has not unary operator.

This requires additional work. So we delayed it for phase 3.

```
long long absv = (v > 0 ? v : (-v));
if (absv > (1LL << 31)) SetError(t, "invalid number.");
// INT_MAX + 1 will be handled when phase 3 : type checking
```

3.2.3 AST node's type

Since the project 3 is about type checking, we don't touch any codes concerning type in skeleton code. So our parser can print incorrect type. we will fix this in phase 3.