Compiler Project #2 - Parser

2012-11269 Dongjae Lim
2013-11403 Chansu Park

1. implementation design

1) `CAstModule*` **`module`**

module = "*module*" *ident* ";" *varDeclaration* {*subroutineDecl* } "*begin*" *statSequence* "*end*" *ident* ".".

We added symbol table and initialize it with predefined functions.

a) `void` **`InitSymbolTable`**

Add predefined I/O-related functions to module's symbol table.

```
- DIM(a: ptrto array; dim: integer): integer
- DOFS(a: ptrto array): integer
- ReadInt(): integer
- WriteInt(i: integer)
- WriteChar(c: char)
- WriteStr(str: char[])
- WriteLn()
```

The others are not significantly different from initial implementation.

2) `CSymtab*` **`varDeclaration`**

varDeclaration = ["*var*" *varDeclSequence* ";"].
varDeclSequence = *varDecl* { ";" *varDecl* }.

`varDeclaration` and `varDeclSequence` both consumes `tSemicolon`, so we decided to not separate `varDelcaration` as a method for avoid ambiguity.
We implemented this with modified definition:

["*var*" *varDeclSequence* ";"]
⇔ ["*var*" *varDecl* { ";" *varDecl* } ";"]
⇔ ["*var*" *varDecl* ";" { *varDecl* ";" }]

Since $FIRST(varDecl) = \{ tIdent \}$, the parser consumes tokens until current token is `tSemicolon` and next token is not `tIdent`.

3) `CSymtab*` **`varDecl`**

varDecl = ident { "," *ident* } " : " *type*.

Since the type of variables in the declaration occurs after all idents are consumed, we have to save names of each ident in other space(we used `vector<string>`), and make symbols after the type has parsed.

Ex. `var a, b, c : integer` → `var_names = vector<string> {“a”, “b”, “c”}`

a) `varDecl` as procedure’s parameter

formalParam = "(" [*varDeclSequence*] ")".

If `varDecl` is called by `formalParam`, we have to save the order of the arguments. But since existing symbol table is implemented as a map, it cannot recognize the order. Therefore we added another ordered array(we used `vector<CSymParam*>`) to control parameters of the subroutine. Also, if the parameter has an array type, we have to make a pointer<array> instead of raw array type.

4) `CAstProcedure*` `subroutineDecl`

subroutineDecl = (*procedureDecl* | *functionDecl*) *subroutineBody ident* ";".
procedureDecl = "*procedure*" *ident* [*formalParam*] ";".
functionDecl = "*function*" *ident* [*formalParam*] " : " *type* ";".
formalParam = "(" [*varDeclSequence*] ")".
subroutineBody = *varDeclaration* "*begin*" *statSequence* "*end*".

Its structure is similar to the structure of the `module`. Since we decided not to separate `varDeclSequence`, we changed the definition of `subroutineDecl` since `varDeclSequence` is used in `formalParam` and `subroutineDecl` includes `formalParam` by `subroutineBody`.

formalParam = "(" [*varDeclSequence*] ")"
⇔ "(" [*varDecl* { ";" *varDecl* }] ")"

Also, in this definition, `formalParam` preserves the order of its parameters. Therefore we edited the prototype of `varDecl` to handle `vector<CSymParam*>` vector as an another parameter. (We already explained about this.) If other functions have to put variables into the symbol table, they will call `varDecl` with `NULL` pointer.

There was another problem. We have to put all parameters into the `CAstProcedure`. These parameters are proceeded by `formalParam`, which is proceeded before the return type of this subroutine is decided. Therefore we have to make `CAstProcedure` object after we get this return type, but if we do not save parameters in the memory, we will lose these. Therefore we made a temporary symbol table to save these parameters into it. After the return type is decided, move the parameter into the symbol table of `CAstProcedure` object all-by-hand.

5) `CAstType*` `ReadType`

type = *basetype* | *type* "[" [*number*] "]".

This definition contains left recursion, so we edited definition into:

type = *basetype type′* | *type* "[" [*number*] "]"
type′ = ε | { "[" [*number*] "]" } *type′*

After we parse entire tokens in the definition, get a proper type symbol through `CTypeManager`. If the type is an array, since we have to take all sizes and make type from the last dimension, read all, push into stack, and solve.
Ex. `integer[5][3]` → `<array 5 of <array 3 of <integer>>>`

6) `CAstStatement*` **`statSequence`**

statSequence = [*statement* { ";" *statement* }].

We separated statement function from the `statSequence` function.
Note that $FIRST(statement) = \{ tIdent, tIf, tWhile, tReturn \}$.

7) `CAstStatement*` **`statement`**

statement = *assignment* | *subroutineCall* | *ifStatement* | *whileStatement* | *returnStatement*.

When the first is `tIf`, `tWhile`, `tReturn`, it is straight-forward.
Otherwise, both `assignment` and `subroutineCall` are the candidates. Then, find this symbol from the given symbol table. If it is found as a procedure, call `subroutineCall`, else call `assignment`.

a) `CAstStatAssign*` **`assignment`**

assignment = *qualident* " := " *expression*.

We implemented `qualident` parser and `expression` parser already(be described below), so we could implemented this simply.

b) `CAstStatCall*` **`subroutineCall`**

subroutineCall = *ident* "(" [*expression* { "," *expression* }] ")".

$FIRST(exprssion) = \{ tPlusMinus, tIdent, tNumber, tBool, tCharacter, tString, tLParen, tNot \}$
If the token next to `tLParen` is not in $FIRST(expression)$, skip reading `expression` and just consume `tRParen`.

c) `CAstStatIf*` **`ifStatement,`** `CAstStatWhile*` **`whileStatement,`** `CAstStatReturn*` **`returnStatement`**

ifStatement = "*if*" "(" *expression* ")" "*then*" *statSequence* ["*else*" *statSequence*] "*end*".
whileStatement = "*while*" "(" *expression* ")" "*do*" *statSequence* "*end*".
returnStatement = "*return*" [*expression*].

It is also straight-forward. `returnStatement` also can consume `expression` or not, so

we applied same method as `subroutineCall`.

8) `CAstExpression*` **`factor`**

factor = *qualident* | *number* | *boolean* | *char* | *string* |
"(" *expression* ")" | *subroutineCall* | "!" *factor*.

If the first token is `tIdent`, `qualident` and `expSubroutineCall`(which has the same EBNF definition as `subroutineCall`) colides. Using similar idea on `varDeclaration`, find this ident from the given symbol table. If it is a procedure, call `expSubroutineCall`, otherwise call `qualident`.

a) `CAstDesignator*` **`qualident`**

qualident = *ident* { "[" *expression* "]" }.

Get the symbol of the consumed `tIdent`'s name, and make a `CAstDesignator` using it. If the array bracket follows, change it into `CAstArrayDesignator`.

b) `CAstFunctionCall*` **`expSubroutineCall`**

It exists to differ from the `subroutineCall` which is called from statement. These two functions differ only in the return type: `CAstFunctionCall*` in this function, `CAstStatCall*` in the function called by statement. Inner structure is the same as `subroutineCall`.

9) Expressions - `CAstExpression*` **`term, simpleexpr, expression`**

term = *factor* { *factOp factor* }.
simpleexpr = ["+" | "−"] *term* { *termOp term* }.
expression = *simpleexpr* [*relOp simpleExpr*].

Hierarchy of expressions are already very well-defined, so we could implemented these without any difficulty.

10) Constants

Constants are very simple, so we just implemented it following its definitions.

a) `CAstConstant*` **`number`**

We added the range checking logic(`-2^31 ~ 2^31`).

b) `CAstConstant*` **`boolean`**

We implemented a string-integer converting function(`strtobool`) to change **"`true`"** and **"`false`"** into a corresponding integer value.

c) `CAstConstant*` **`character`**

The `tCharacter` token has a character, but we need to handle this as an integer. So we

took the value of the token and cast it into integer value.

d) `CAstStringConstatnt*` **`stringConst`**

There was an error in default AST's implementation - `CAstStringConstant::GetType()` always returns `NULL`. So we fixed it to return correct type(array of char).

2. Differences between our parser and the reference parser

1) Type Checking

We just adapted the formula of the type. If the reference parser parses the type as a pointer, we also parse it as a pointer. Also, if it is an array with dimension n, our parser also parses it as dimension n array type. However, we don't check some properties in this step.

a) Open array

We don't decide the open array's size.
Ex. In the test/parser/array.mod - `addB` parameter (`var A, B, C: integer[][]`)
Reference_parser: `<ptr(4) to <array 5 of <array 5 of <int>>>>`
(addB is called only with `integer[5][5]` arrays as its parameters.)
Test_parser: `<ptr(4) to <array of <array of <int>>>>`
(we don't know about the size until we analyze it semantically.)

b) INVALID types / wrong types

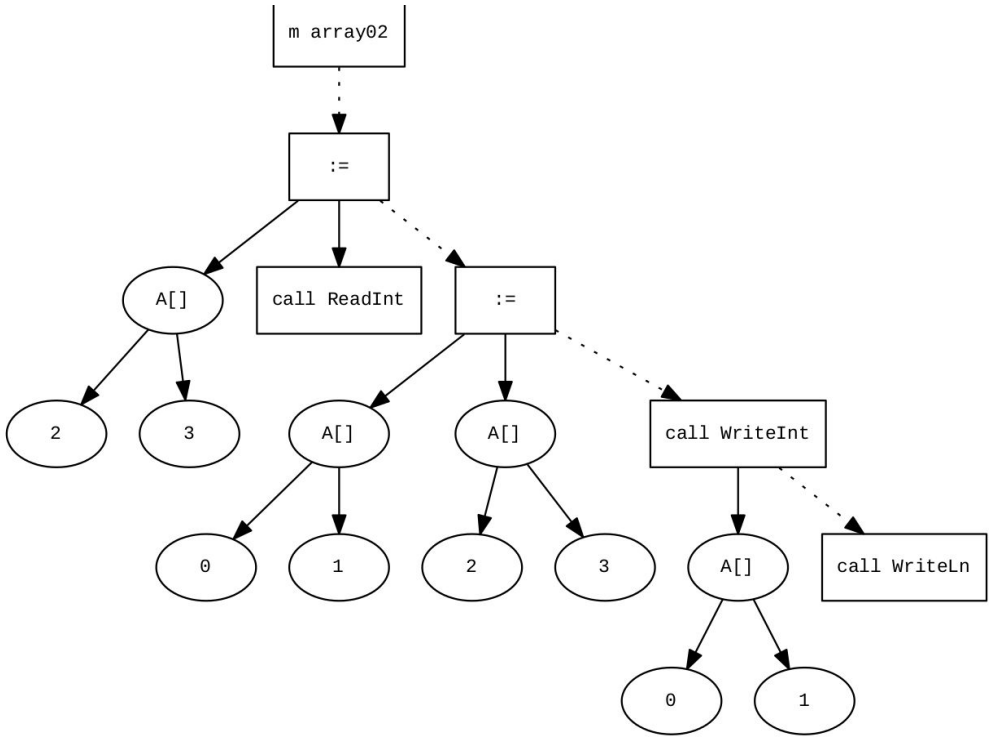
In this step, we don't have to match types using type inference rules. So many of the variables have wrong or invalid types, but we don't care.

2) referencing

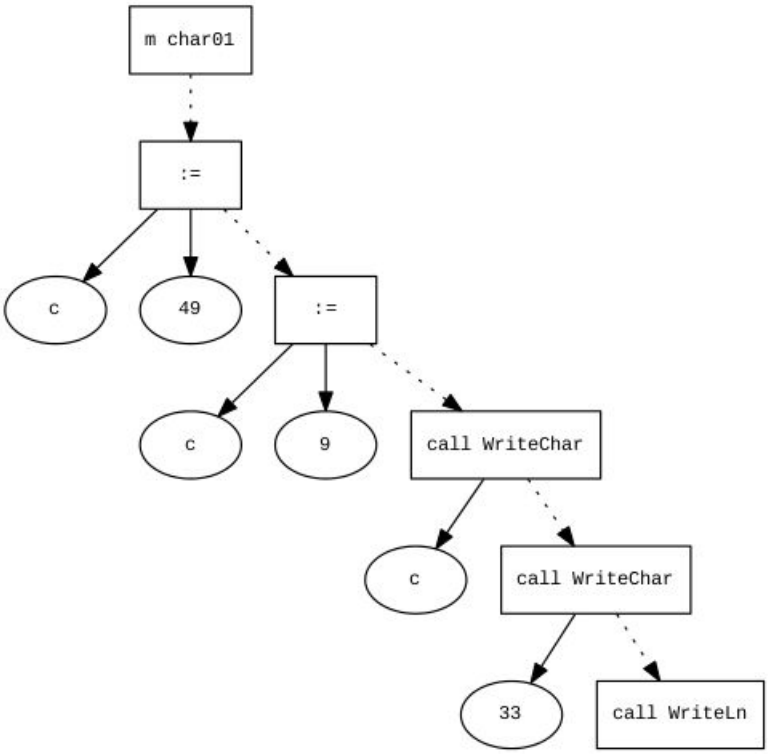
In the `subroutineCall`, if we call with the array-type parameters. we have to send a reference to these parameters according to the reference parser, but we also don't have to do that in this step.
Also in the reference parser, strings are always included by a pointer to that array of character. But we don't implemented it in this step.

3. Some ASTs

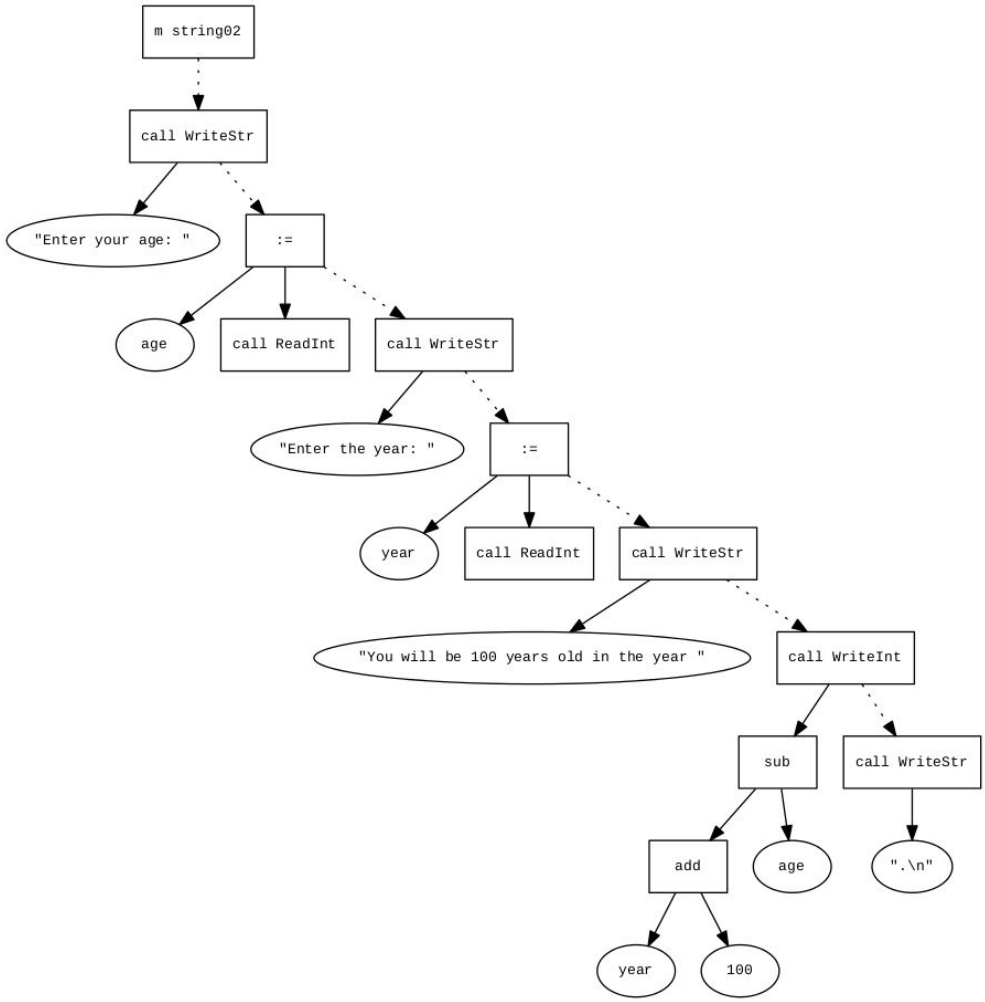
Array02:



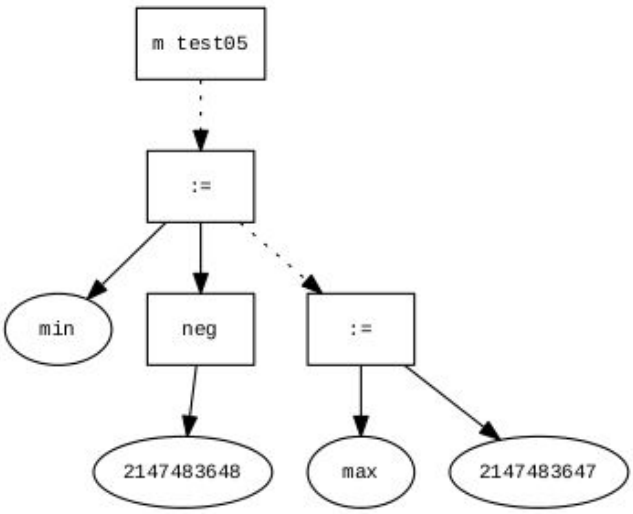
Char01:



string02:



Test05:



We also uploaded other ASTs in our github reposiroty.