Syntax Analysis

Sungbin Jo (2021-13630), Kijun Shin (2021-15391)

CppSnuPL implementation notes

The provided skeleton code is opinionated and detailed, so implementing the parser was quite straightforward. Most of the time was spent experimenting with the SwiftSnuPL implementation, so frankly nothing is quite interesting or notable from our current implementation.

Implemented methods

Every significant production rule from the provided SnuPL/2 EBNF is translated to each corresponding method.

```
class CParser {
  /// @name methods for recursive-descent parsing
  /// @{

  CAstModule *module(void);

  void constDeclSequence(CAstScope *s);
  void varDeclSequence(CAstScope *s);
  const CType *varDecl(CAstScope *s, vector<string> *idents);

  void subrDeclaration(CAstScope *s);
  CAstProcedure *procedureDecl(CAstScope *s);
  CAstProcedure *functionDecl(CAstScope *s);
  void formalParam(CAstScope *s, vector<CSymParam *> *params);

  CAstStatement *statSequence(CAstScope *s);

  CAstStatAssign *assignment(CAstScope *s);
  CAstStatIf *ifStatement(CAstScope *s);
  CAstStatWhile *whileStatement(CAstScope *s);
  CAstStatReturn *returnStatement(CAstScope *s);

  CAstFunctionCall *subroutineCall(CAstScope *s);

  CAstExpression *expression(CAstScope *s);
```

```
  CAstExpression *simpleexpr(CAstScope *s);
  CAstExpression *term(CAstScope *s);
  CAstExpression *factor(CAstScope *s);

  CAstDesignator *qualident(CAstScope *s);

  CAstConstant *number(void);
  CAstConstant *boolean(void);
  CAstConstant *charConst(void);
  CAstStringConstant *stringConst(CAstScope *s);

  const CType *type(CAstScope *s);
  const CType *basetype(void);

  /// @}
}
```

Handling implicit array-pointer conversion

Every array-typed parameter in parameters are passed as pointers. The Cpp-SnuPL handles these conversions during parsing and parses array parameters as pointers; an implicit addressof (the `&` special operation) is inserted by the parser when passing arrays to subroutines.

Notable differences from the provided code

Parser has two token lookahead

To distinguish between an assignment and a subroutine call or between an array subscript and a subroutine call, the parser must know the token after the leading identifier. The suggested design was to pass the first token to each parsing function, but it felt quite hackish.

Instead, we saw that the parser already had an interal `_token` field that isn't used anywhere in the skeleton code. We modified the `Consume()` method to stash the new token to the `_token`, updated `Peek()` and created `PeekNext()` appropriately.

SwiftSnuPL: An experimental Swift port of SnuPL

Motivation

The unfamiliarity on the C++ language pushed us to experiment with porting the compiler to a more familiar language.

Partially porting the parser to Swift showed a significant productivity increase not only from familiarity but also from the conciseness of the language, algebric data types, and pattern matching.

We decided to use the Swift port as a sandbox for experimenting with various design deviations from the original design.

SwiftSnuPL design notes

Below are some parser design deviations from the original skeleton code:

- The AST produced by the parser is a dumb interpretation of the provided code.
  - Any implicit casts or conversions are not represented in the AST.
  - The parser doesn't have knowledge on symbol resolution or types.
- Symbol resolution and type checking is done in a resolve stage separate from parsing, and produces a side table outside of the AST that contains symbol and type information.
  - We're not confident with this design decision; we are not sure if this will play out as expected when implementing the TAC generator.

SwiftSnuPL design and implementation notes

Model the AST as dumb sum types instead of subclasses

In our opinion, this encodes more naturally in code. This also allows separating logic between stages are easier: we can pattern match the AST nodes outside of the AST implementation, instead of relying on the langauge's dynamic dispatch. (Arguably this is just as possible with implementing something like a visitor pattern, but this is even more verbose.)

- Have the types `Expression`, `Statement`, `Type`, `Declaration` as a sum type (enumerations with associated values in Swift speech)
  - The target of an `.assignment` is an `Expression` for future extensions. (The parser only parses targets)
- Generates a complete parse tree without considering any

Reuse of the CppSnuPL scanner

Thanks to the Swift's C interop feature, we can reuse the C++ implementation of the scanner. The SPM target `CppSnuPL` and `SwiftSnuPL/Scanner.swift` implements the scanner.

Off-topic, but this feature was pretty cool: throw in some C header files, and the embbedded clang in the Swift compiler can transparently import functions.

Testing by pretty-printing

We have added a pretty printer PoC that allows regenerating compatible SnuPL code from the parsed module. We've found some parser bugs by parsing test cases, pretty printing, reparsing the code and comparing the results.

- Have implemented a complete `Parser` and `Resolver`, a `TACGenerator` PoC is also implemented.
- Added a pretty printer PoC that allows regenerating compatible SnuPL code
  - Testing the parser by parsing, pprinting and reparsing, and comparing the results

Implemented types and grammar of `Parser`

Below are the Swift types used to represent the AST.

Most are boring; one interesting point is that the AST represents subroutines in calls and assignment targets as a generic `Expression` instead of an identifier. The parser currently rejects any code that doesn't call a single identifier (and the resolver rejects while type checking as well), but future language extensions might allow more expressions (like an item of array of functions) to be callable as well.

```
indirect enum Expression {
    case unary(`operator`: Token, value: Expression)
    case binary(`operator`: Token, `left`: Expression, `right`: Expression)
    case `subscript`(array: Expression, index: Expression)
    case call(function: Expression, arguments: [Expression])
    case variable(name: Token)
    case integer(Int32)
    case longint(Int64)
    case boolean(Bool)
    case char(UInt8)
    case string([UInt8])
}

indirect enum Statement {
    case assignment(target: Expression, value: Expression)
    case call(procedure: Expression, arguments: [Expression])
    case `if`(condition: Expression, thenBody: [Statement], elseBody: [Statement])
    case `while`(condition: Expression, body: [Statement])
    case `return`(value: Expression?)
}

indirect enum `Type` {
    case boolean
    case char
    case integer
    case longint
    case array(base: `Type`, size: Expression?)
}
```

```swift
struct Parameter {
    let name: Token
    let type: `Type`
}

indirect enum Declaration {
    case `var`(name: Token, type: `Type`)
    case `const`(name: Token, type: `Type`, initializer: Expression)
    case procedure(name: Token, parameters: [Parameter], block: Block?)
    case function(name: Token, parameters: [Parameter], `return`: `Type`, block: Block?)
}

struct Module {
    let name: Token
    let block: Block
}

struct Block {
    let declarations: [Declaration]
    let body: [Statement]
}
```

Below is the equivalent SnuPL grammar as implemented in the Swift parser. Comments indicate the parser function that implemented the associated grammar.

```
(* parseModule() *)
module = "module" ident ";"
         { constDeclarations | varDeclarations | functionDeclaration | procedureDeclaration
         [ "begin" statements ] "end" ident "." ;

(* Declarations *)
(* parseVarDeclarations() -> [Declaration] *)
varDeclarations = "var" varDeclaration ";" { varDeclaration ";" } ;
varDeclaration = ident { "," ident } ":" type ;

(* parseConstDeclarations() -> [Declaration] *)
constDeclarations = "const" constDeclaration ";" { constDeclaration ";" } ;
constDeclaration = ident { "," ident } ":" type "=" expression ;

(* parseFunctionDeclaration() -> Declaration *)
functionDeclaration = "function" ident [ parameters ] ":" type ";"
                      ( "extern" | functionBody ident ) ";" ;
functionBody = [ constDeclarations ] [ varDeclarations ] "begin" statements "end" ;

(* parseProcedureDeclaration() -> Declaration *)
```

```
procedureDeclaration = "procedure" ident [ parameters ] ";"
                       ( "extern" | procedureBody ident ) ";" ;
procedureBody = [ constDeclarations ] [ varDeclarations ] "begin" statements "end" ;

(* parseParameters() -> [Parameter] *)
parameters = "(" parameterDeclarations ")" ;
parameterDeclarations = [ parameterDeclaration { ";" parameterDeclaration } ] ;
parameterDeclaration = ident { "," ident } ":" type ;

(* Statements *)
statements = [ statement { ";" statement } ] ;                                    (* parseStat
statement = assignment | procedureCall | if | while | return ;                    (* parseStat
assignment = subscript ":=" expression ;                                          (* parseAssi
procedureCall = ident "(" [ expression { "," expression } ] ")" ;                 (* parsePro
if = "if" "(" expression ")" "then" statements [ "else" statements ] "end" ;      (* parseIf()
while = "while" "(" expression ")" "do" statements "end" ;                        (* parseWhi
return = "return" [ expression ] ;                                                (* parseRetu

(* Types *)
type = baseType { "[" [ plusMinus ] "]" } ;             (* parseType() -> `Type` *)
baseType = "boolean" | "char" | "integer" | "longint" ; (* parseBaseType() -> `Type` *)

(* Expressions *)
expression = relation ;                                                              (* parse
relation = plusMinus [ ( "=" | "#" | "<" | "<=" | ">" | ">=" ) plusMinus ] ;        (* parse
plusMinus = [ "+" | "-" ] mulDiv { ( "+" | "-" | "||" ) mulDiv } ;                   (* parse
mulDiv = primary { ( "*" | "/" | "&&" ) primary } ;                                  (* parse
primary = subscript | literal | "(" expression ")" | functionCall | "!" primary ;   (* parse
literal = number | boolean | char | string ;                                         (* parse
subscript = variable [ "[" plusMinus "]" ] ;                                         (* parse
functionCall = variable "(" [ expression { "," expression } ] ")" ;                  (* parse
variable = ident ;                                                                   (* parse
```

Implementation notes on `Resolver`

FYI to the reader: the `Resolver.swift` is not yet committed to the repository at the time of writing. It will be pushed soon.

The resolver resolves symbols and type-checks the code. It maintains a separate side table with a type mapping for each subexpression in the parse tree, and a symbol mapping for each token that represents a variable.

The resolver introduces new types to represent types, symbols and scopes as below.

```
indirect enum `Type` {
    case boolean
```

```
    case char
    case integer
    case longint
    case array(base: `Type`?, size: Int32?)
    case procedure(parameters: [`Type`])
    case function(parameters: [`Type`], `return`: `Type`)

    var isScalar: Bool { /* ... */ }
    func isAssignable(to target: Self) -> Bool { /* ... */ }
    func isConvertible(to target: Self) -> Bool { /* ... */ }
}

enum Symbol {
    case `var`(token: Token, type: `Type`)
    case const(token: Token, type: `Type`, initializer: AnyHashable)
}

struct Scope {
    var symbols: Set<Symbol>
    let `return`: `Type`?

    mutating func addVar(token: Token, type: `Type`) { /* ... */ }
    mutating func addConst(token: Token, type: `Type`, initializer: AnyHashable) { /* ... */
    func findSymbol(named name: String) -> Symbol? { /* ... */ }
}
```

The resolver uses a separate type different from `Parser.Type` for representing internal types, including procedure and function types as well as an array without type restrictions (anyarray), used for the `DIMS` and `DOFS` internal functions (anyarrays are represented as `.array(base: nil, size: nil)`).