TitoCC - C-kääntäjä ttk-91 -arkkitehtuurille.

T.A.

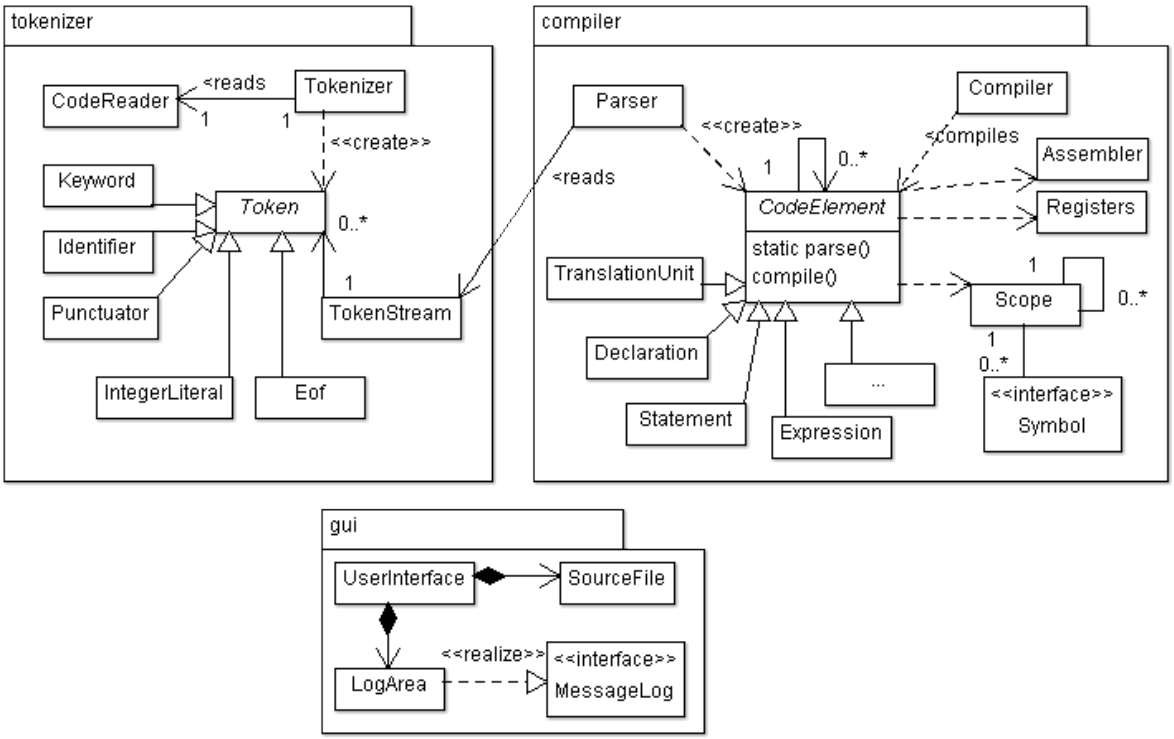
11. tammikuuta 2013

Rakenne ja yleinen toiminta

Käännös suoritetaan kolmessa osassa. Ensin lähdekoodi analysoidaan tokenisoijan avulla, joka jakaa sen pienimpiin mahdollisiin palasiin, eli välimerkkeihin, tunnisteisiin, vakioihin, ja avainsanoihin.

Tämän jälkeen parseri muodostaa token-listasta puurakenteen, jossa jokaista koodielementtiä vastaan oma `CodeElement`-tyyppinen objekti. Lopuksi kääntäjä käy puurakenteen rekursiivisesti läpi ja generoi niitä vastaavat käskyt. Se myös tarkistaa muun muassa, että jokaisen elementin tyyppi on oikea, ja että kaikki viitatut muuttujat ovat olemassa.

Kuva 1: Luokka- ja pakettikaavio.



Symbolien hallinta

Käännöksen aikana pidetään kirjaa kaikista esitellyistä symboleista erillisessä `Scope`-hierarkiassa, joka määrittelee jokaiselle symbolille oman näkyvyysalueensa. Symboleita on neljää tyyppiä: muuttujat (`VariableDeclaration`), parametrit (`Parameter`), funktiot (`Function`) ja kääntäjän sisäiseen käyttöön tarkoitetut symbolit (`InternalSymbol`).

Rekisterien allokointi

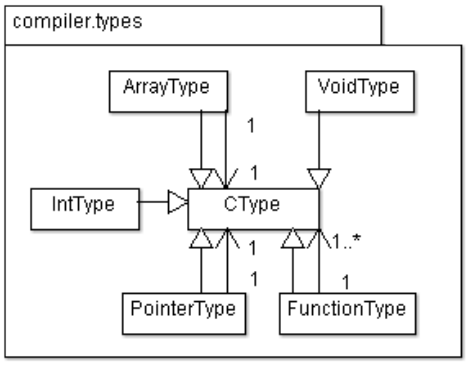
Käännöksen aikana ylläpidetään eräänlaista pinoa kullakin hetkellä käytetyistä rekistereistä. Tämä on toteutettu `Registers`-luokassa.

Rekisteripinon huipulla ovat ns. aktiiviset rekisterit, joita tarvitaan sen hetkisen operaation suorittamiseen. Jokaisen lausekkeen (`Expression`) evaluointi palauttaa arvonsa ensimmäisessä aktiivisessa rekisterissä. Aktiivisia rekistereitä voidaan tarvittaessa allokoida lisää, ja aktiivisen alueen alusta voidaan poistaa rekistereitä alilausekkeiden evaluoimiseksi. Rekisteripinon alussa olevia rekistereitä voiaan kierrättää, eli ne voidaa tallentaa väliaikaisesti muistiin, jolloin ne ovat vapaita uudelleenallokoitavaksi.

Tyyppijärjestelmä

Osoittimien, taulukoiden ja tyyppitarkistusten toteuttamiseksi on abstraktoitu C:n tyyppijärjestelmä omaan luokkahierarkiaansa. Sen avulla voidaan vertailla tyyppejä keskenään ja tiedustella mitä operaatioita kukin tyyppi tukee. Jokaisella `Expression`-objektilla on oma paluuarvotyyppinsä, joka voi riippua kyseisen lausekkeen operandien tyypeistä. Myös jokaisella symbolilla on oma tyyppinsä.

Kuva 2: Tyyppijärjestelmä.



Kielen EBNF-määrittely

Tokenisoijan tunnistama syntaksi voidaan määritellä EBNF-notaation avulla:

```
TOKENIZER_SYNTAX = {{WHITESPACE} TOKEN} {WHITESPACE} EOF
TOKEN = PUNCTUATOR | IDENTIFIER | KEYWORD | INTEGER_LITERAL
PUNCTUATOR_TOKEN = "+" | "++" | "+=" | "-" | "--" | "-=" | "*" | "*=" | "/"
        | "/=" | "%" | "%=" | "|" | "|=" | "||" | "&" | "&=" | "&&" | "~"
        | "~=" | "^" | "^=" | "!" | "=" | ">>" | ">>=" | "<<" | "<<=" | "=="
        | "!=" | "<" | "<=" | ">" | ">=" | "{" | "}" | "[" | "]" | "("
        | ")" | ";" | ","
INTEGER_LITERAL = DIGIT {DIGIT} WORD
IDENTIFIER = WORD - KEYWORD
KEYWORD = "while" | "if" | "else" | "return" | "int" | "void"
WORD = IDENTIFIER_START_CHAR {IDENTIFIER_CHAR}
IDENTIFIER_START_CHAR = "_" | ASCII_LETTER
IDENTIFIER_CHAR = "_" | ASCII_LETTER | DIGIT
WHITESPACE = ? space, tab, newline etc ?
ASCII_LETTER = ? "a" ... "z", "A" ... "Z" ?
DIGIT = ? "0" ... "9" ?
```

Ja vastaavasti parserin tunnistamat koodielementit:

```
TRANSLATION_UNIT = {DECLARATION} EOF
DECLARATION = VARIABLE_DECLARATION | FUNCTION
VARIABLE_DECLARATION = TYPE_SPECIFIER DECLARATOR ["=" EXPRESSION] ";"
DECLARATOR = "*" DECLARATOR | DIRECT_DECLARATOR
DIRECT_DECLARATOR = IDENTIFIER | "(" DECLARATOR ")" |
        DIRECT_DECLARATOR "[" EXPRESSION "]"
FUNCTION = TYPE_SPECIFIER IDENTIFIER PARAMETER_LIST BLOCK_STATEMENT
PARAMETER_LIST = "(" [PARAMETER {"," PARAMETER}] ")"
PARAMETER = TYPE_SPECIFIER DECLARATOR
BLOCK_STATEMENT = "{" {STATEMENT} "}"
STATEMENT = EXPRESSION_STATEMENT | DECLARATION_STATEMENT |
        IF_STATEMENT | WHILE_STATEMENT | BLOCK_STATEMENT |
        RETURN_STATEMENT | ";"
EXPRESSION_STATEMENT = EXPRESSION ";"
DECLARATION_STATEMENT = VARIABLE_DECLARATION
IF_STATEMENT = "if" "(" EXPRESSION ")" STATEMENT ["else" STATEMENT]
WHILE_STATEMENT = "while" "(" EXPRESSION ")" STATEMENT
RETURN_STATEMENT = "return" [EXPRESSION] ";"
TYPE_SPECIFIER = "void" | "int"
EXPRESSION = ASSIGNMENT_EXPRESSION
ASSIGNMENT_EXPRESSION = BINARY_EXPRESSION (ASSIGNMENT_OPERATOR
        ASSIGNMENT_EXPRESSION)?
ASSIGNMENT_OPERATOR = "=" | "+=" | "*=" | "&=" | "|=" | "^=" | "-=" |
        "/=" | "%=" | "<<=" | ">>="
BINARY_EXPRESSION = [BINARY_EXPRESSION "||"] BINARY_EXPRESSION2
BINARY_EXPRESSION2 = [BINARY_EXPRESSION2 "&&"] BINARY_EXPRESSION3
BINARY_EXPRESSION3 = [BINARY_EXPRESSION3 "|"] BINARY_EXPRESSION4
BINARY_EXPRESSION4 = [BINARY_EXPRESSION4 "^"] BINARY_EXPRESSION5
```

```
BINARY_EXPRESSION5 = [BINARY_EXPRESSION5 "&"] BINARY_EXPRESSION6
BINARY_EXPRESSION6 = [BINARY_EXPRESSION6 "=="] BINARY_EXPRESSION7
BINARY_EXPRESSION7 = [BINARY_EXPRESSION7 "!=") BINARY_EXPRESSION8
BINARY_EXPRESSION8 = [BINARY_EXPRESSION8 ("<" | "<=" | ">" | ">=")]
        BINARY_EXPRESSION9
BINARY_EXPRESSION9 = [BINARY_EXPRESSION9 ("<<" | ">>")] BINARY_EXPRESSION10
BINARY_EXPRESSION10 = [BINARY_EXPRESSION10 ("+" | "-")] BINARY_EXPRESSION11
BINARY_EXPRESSION11 = [BINARY_EXPRESSION11 ("*" | "/" | "%")] PREFIX_EXPRESSION
PREFIX_EXPRESSION = ("++" | "--" | "+" | "-" | "!" | "~") PREFIX_EXPRESSION |
        POSTFIX_EXPRESSION
POSTFIX_EXPRESSION = POSTFIX_EXPRESSION ("++" | "--") |
        INTRINSIC_CALL_EXPRESSION | FUNCTION_CALL_EXPRESSION |
        SUBSCRIPT_EXPRESSION | PRIMARY_EXPRESSION
FUNCTION_CALL_EXPRESSION = POSTFIX_EXPRESSION ARGUMENT_LIST
INTRINSIC_CALL_EXPRESSION = ("in" | "out") ARGUMENT_LIST
SUBSCRIPT_EXPRESSION = POSTFIX_EXPRESSION "[" EXPRESSION "]"
ARGUMENT_LIST = "(" [EXPRESSION {"," EXPRESSION}] ")"
PRIMARY_EXPRESSION = IDENTIFIER_EXPRESSION | INTEGER_LITERAL_EXPRESSION |
        "(" EXPRESSION ")"
IDENTIFIER_EXPRESSION = IDENTIFIER
```