FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

G-assignment in Compilers

Simon Shine, Kristoffer Søholm, Mathias Svensson

A Compiler for Janus

December 16th, 2009

Simon Shine
Kristoffer Søholm
Mathias Svensson

Contents

Simon Shine
Kristoffer Søholm
Mathias Svensson

Introduction

Our additions to the partially complete compiler are divided into four files, `Parser.grm`, `Lexer.lex`, `Type.sml` and `Compiler.sml`. This report documents our additions in chapters corresponding to this division.

Simon Shine
Kristoffer Søholm
Mathias Svensson

Lexer

The missing lexical tokens were generated either directly by regular expression rules (`[`, `]`, `==`, `<`, `!`, `&&` and `||`), or as return values from the `keyword` function that gathers all possible tokens matching the regular expression `[a-zA-Z][a-zA-Z0-9]*`. This simplifies separation of keywords from identifiers which otherwise share the same namespace. Keywords thus get precedence over identifiers of similar names, making them "reserved".

Simon Shine
Kristoffer Søholm
Mathias Svensson

Parser

Tokens were added with their corresponding type. Token types have constructors `of int*int` except for tokens `NUM` and `ID` that are `of int*(int*int)` and `of string*(int*int)`, respectively. These numbers denote position (line, column) for debugging purposes.
Ambiguity is resolved through operator precedence cf. [?, ch. 18.2.2, p. 33]. That is, precedence between syntactic elements of same associativity is handled by the order in which they are listed, highest precedence at the bottom.
Productions described in the Janus grammar are added. In particular, the following:

- Array definitions (for declaring array variables as input, intermediate and output):

```
Defs : ID Defs      { Janus.IntVarDef $1 :: $2 }
     | ID LBRACK NUM RBRACK Defs
                    { Janus.ArrayVarDef (#1 $1, #1 $3, #2 $1) :: $5 }
     |              { [] }
```

- Calling and uncalling statements:

```
Stat : Stat SEMICOLON Stat  { Janus.Sequence ($1, $3, $2) }
     | Lval ADD Exp         { Janus.AddUpdate ($1, $3, $2) }
     | Lval SUBTRACT Exp    { Janus.SubUpdate ($1, $3, $2) }
     | SKIP                 { Janus.Skip $1 }
     | CALL ID              { Janus.Call (#1 $2, $1) }
     | UNCALL ID            { Janus.Uncall (#1 $2, $1) }
     | IF Cond THEN Stat ELSE Stat FI Cond
                            { Janus.If ($2, $4, $6, $8, $1) }
     | FROM Cond DO Stat LOOP Stat UNTIL Cond
                            { Janus.Loop ($2, $4, $6, $8, $1) }
```

- Left-side values for assignments (using the `+=` and `-=` operators):

```
Lval : ID                   { Janus.IntVar $1 }
     | ID LBRACK Exp RBRACK { Janus.ArrayIndex(#1 $1, $3, #2 $1) }
```

- Conditions:

```
Cond : Exp LESS Exp     { Janus.Less($1, $3, $2) }
     | Exp EQ Exp       { Janus.Equal($1, $3, $2) }
     | NOT Cond         { Janus.Not($2, $1) }
     | Cond AND Cond    { Janus.And($1, $3, $2) }
     | Cond OR Cond     { Janus.Or($1, $3, $2) }
     | LPAR Cond RPAR   { $2 }
```

The parser-syntactic methods used here are primarily: using `$1`, `$2`, ... to refer to enumerated tokens of a production and `#1`, `#2`, ... to refer to the Standard ML polymorphic functions. Also, constructing values recursively by referring to an enumeration that points to a non-terminal (as seen in `Defs`.)

Simon Shine
Kristoffer Søholm
Mathias Svensson

Type check

$Check_{Exp}$

```
checkExp(Exp, vtable, avoid) = case Exp of
    num => ok
  | id  => v := lookup(vtable, name(id))
           if type(v) is not integer
             then error (Array variable used as integer)
             else
           if name(v) = avoid
             then error (LHS variable used on RHS)
             else ok
  | id ‘[‘ Exp1 ‘]‘ =>
           checkExp(Exp1, vtable, ftable, avoid)
           v := lookup(vtable, name(id))
           if type(v) is not array
             then error (Integer variable used as array)
             else
           if name(v) = avoid
             then error (LHS variable used on RHS)
             else ok
  | Exp1 ‘+‘ Exp2
  | Exp1 ‘-‘ Exp2 =>
           checkExp(Exp1, vtable, avoid)
           checkExp(Exp2, vtable, avoid)
  | Exp1 ‘/2‘ =>
           checkExp(Exp1, vtable, avoid)
```

$Check_{Stat}$

```
checkStat(Stat, vtable, pnames) = case Stat of
    Stat1 ‘;‘ Stat2 =>
           checkStat (Stat1, vtable, pnames)
           checkStat (stat2, vtable, pnames)
  | id ‘+=‘ Exp2
  | id ‘-=‘ Exp2 =>
           v := lookup(vtable, name(id))
           if type(v) is not integer
             then error (Array variable used as integer)
             else checkExp(Exp2, vtable, name(id))
  | id ‘[‘ Exp1 ‘]‘ ‘+=‘ Exp2
  | id ‘[‘ Exp1 ‘]‘ ‘-=‘ Exp2 =>
           checkExp(Exp1, vtable, none)
           v := lookup(vtable, name(id))
           if type(v) is not integer
             then error (Integer variable used as an array)
             else checkExp(Exp2, vtable, name(id))
  | ‘if‘ Cond1 ‘then‘ Stat1 ‘else‘ Stat2 ‘fi‘ Cond2 =>
           checkCond(Cond1, vtable)
           checkStat(Stat1, vtable, pnames)
```

```
           checkCond(Cond2, vtable)
           checkStat(Stat2, vtable, pnames)
  | ‘from‘ Cond1 ‘do‘ Stat1 ‘loop‘ Stat2 ‘until‘ Cond2 =>
           checkCond(Cond1, vtable)
           checkStat(Stat1, vtable, pnames)
           checkCond(Cond2, vtable)
           checkStat(Stat2, vtable, pnames)
  | Skip => ok
  | Call pname
  | Uncall pname =>
           v := lookup(pnames, pname)
           if v is unbound
             then error (Unknown procedure: pname)
             else ok
```

$Check_{Cond}$

```
checkCond(Cond, vtable) = case Cond of
    Exp1 ‘==‘ Exp2
  | Exp1 ‘<‘  Exp2 =>
           checkExp(Exp1, vtable, none)
           checkExp(Exp2, vtable, none)
  | ‘!‘ Cond1 => checkCond(Cond1, vtable)
  | Cond1 ‘&&‘ Cond2
  | Cond1 ‘||‘ Cond2 =>
           checkCond(Cond1, vtable)
           checkCond(Cond2, vtable)
```

$Check_{Defs}$

```
checkDefs(Defs, vtable) =
           for each Def in Defs
             v := lookup(name(Def), vtable)
             if v is bound
               then error (Multiple declarations of: name(v))
               else
             if type(v) is array and size = 0
               then error (Zero-sized array: name(v))
               else ok
```

Simon Shine
Kristoffer Søholm
Mathias Svensson

Compiler

It iz so fun!