Escuela de Ingeniería Informática
Universidad de Oviedo

PROGRAMMING LANGUAGE DESIGN

Documentation

Sergio Mosquera Dopico
Uo244835

1. Extensions

- **Increment (++), decrement (--) and arithmetical assignment (+=, -=, *=, /=):** this extension can be achieved using syntax sugar. The main idea is to understand these constructions as assignments e.g. i++ → i=i+1 || i*=6 → i=i*6.
- **For statement:** is not the for each construction. For this statement, we need an initialization (which will be an assignment) that must be executed once; a condition (which will be a logical expression) that must be evaluated on each iteration; and an increment (which will also be an assignment) that must be executed on each iteration. Then we will have a list of statements that will be executed if the condition fulfills.
- **Do while:** a simple construction which evaluates the condition just after executing the list of statements. In this case, we just need a to declare a label while in the 'while' instruction we needed 2 labels. This construction has an expression to be evaluated (must be logical) and if the condition fulfills, the list of statements will be executed again.
- **Multiple assignment and assignment as Lvalue:** to achieve this, we need the assignment to be also an expression. As we say it's an Lvalue, we need to define its code template for the address. With this extension, we can afford assignments like these a=b=4 || a= (b=3) =5.
- **Ternary operator:** the same functionality as in the rest of programming languages. This expression is composed by other three expressions, the first one is the condition to be evaluated (must be logical); the second and the third one are the expression returned if the condition is true or false, respectively.
- **Switch case:** this statement is a bit complex because it must have at least one case, and cases can be normal cases or a default case, which can appear just once. Apart from that, cases can have a break instruction or not, this instruction which will stop evaluating the following cases. The condition for each case must promote to the type of the condition in the switch clause.

- **Passing as reference:** now the compiler allows structs and arrays to be parameters in a function definition. This can be achieved by passing the address of this constructions instead of its value. To know if an invocation is correct, we must check two things on each case:
    - **Array:**
        - The size of the both arrays must be the same.
        - The type of both arrays must be the same.
    - **Struct:**
        - The number of the fields is the same.
        - The fields have the same type (and the same order).

2. Abstract Grammar

2.1 Definition

Program: Program → Definition*

FunctionDefinition: Definition → Type Statement*

VariableDefinition: Definition → Type ID

2.2 Expression

Arithmetic: $Expression_1$ → $Expression_2$ (+|-|*|/) $Expression_3$

Cast: $Expression_1$ → Type $Expression_2$

CharLiteral: Expression → CHAR_CONSTANT

Comparison: $Expression_1$ → $Expression_2$ (==|!=|<|<=|>|>=) $Expression_3$

FieldAccess: $Expression_1$ → $Expression_2$ $Expression_3$

Indexing: $Expression_1$ → $Expression_2$ $Expression_3$

IntLiteral: Expression → INT_CONSTANT

Logical: $Expression_1$ → $Expression_2$ (‘&&’|’||’) $Expression_3$

RealLiteral: Expression → REAL_CONSTANT

Ternary: $Expression_1$ → $Expression_2$ $Expression_3$ $Expression_4$

UnaryMinus: $Expression_1$ → $Expression_2$

UnaryNot: $Expression_1$ → $Expression_2$

Variable: Expression → ID

2.3 Statement

Assignment: $Expression_1$ → $Expression_2$ $Expression_3$

BreakInstruction: Statement

DefaultCase: Statement → Statement*

DoWhileStatement: Statement → Statement* Expression

ForStatement: Statement → $Expression_1$ $Expression_2$ $Expression_3$ Statement*

IfStatement: Statement → Expression $Statement_1$* $Statement_2$*

Invocation: $Expression_1$ → $Expression_2$ Expression*

NormalCase: Statement → Expression Statement*

Read: Statement → Expression*

Return: Statement → Expression

SwitchCase: Statement → Expression Statement*

WhileStatement: Statement → Expression Statement*

Write: Statement → Expression*

2.4 Type

ArrayType: $Type_1$ → $Type_2$ INT_CONSTANT

CharType: Type

ErrorType: Type

FunctionType: Type → Type ID Definition*

IntType: Type

RealType: Type

RecordField: Type → Type ID

Struct: Type → ID Type*

VoidType: Type

3. Code templates

3.1 Address

ADDRESS [Assignment: Expression$_1$ → Expression$_2$ Expression$_3$] () =
  ADDRESS [Expression$_2$] ()
  <DUP> Expression$_2$.Type
  VALUE [Expression$_3$] ()
  CG.ConvertTo(Expression$_3$.Type, Expression$_2$.Type)
  <STORE> Expression$_2$.Type

ADDRESS [FieldAccess: Expression$_1$ → Expression$_2$ Expression$_3$] () =
  ADDRESS [Expression$_2$] ()
  <PUSHI> Expression$_2$.Type.getField(Expression$_1$.Name).Offset
  <ADDI>

ADDRESS [Indexing: Expression$_1$ → Expression$_2$ Expression$_3$] () =
  ADDRESS [Expression$_2$] ()
  VALUE [Expression$_3$] ()
  CG.ConvertTo(Expression$_3$.Type, IntType.getInstance())
  <PUSHI> Expression$_2$.Type.NoB
  <MULI>
  <ADDI>

ADDRESS [Variable: Expression → ID] () =
  If(Expression.Definition.Scope){
    <PUSHA> BP
    <PUSHI> Expression.Definition.Offset
    <ADDI>
  }
  Else{
    <PUSHA> Expression.Definition.Offset
  }
  If (Expression.Definition.Type.isReference){
    <LOADI>
  }

3.2 Value

VALUE [Arithmetic: $Expression_1$ → $Expression_2$ (+|-|*|/) $Expression_3$] () =
  VALUE [$Expression_2$] ()
  CG.ConvertTo($Expression_2$.Type, $Expression_1$.Type)
  VALUE [$Expression_3$] ()
  CG.ConvertTo($Expression_3$.Type, $Expression_1$.Type)
  CG.Arithmetic($Expression_1$, $Expression_1$.Operator)

VALUE [Assignment: $Expression_1$ → $Expression_2$ $Expression_3$] () =
  ADDRESS [$Expression_1$] ()
  <LOAD> $Expression_1$.Type

VALUE [Cast: $Expression_1$ → Type $Expression_2$] () =
  VALUE[$Expression_2$] ()
  CG.ConvertTo($Expression_2$, Type)

VALUE[CharLiteral: Expression → CHAR_CONSTANT] () =
  <PUSHB> CHAR_CONSTANT

VALUE[Comparison: $Expression_1$ → $Expression_2$ (==|!=|<|<=|>|>=) $Expression_3$] () =
  VALUE[$Expression_2$] ()
  CG.ConvertTo($Expression_2$.Type, $Expression_2$.Type.higherThan($Expression_3$.Type))
  VALUE[$Expression_3$] ()
  CG.ConvertTo($Expression_3$.Type, $Expression_2$.Type.higherThan($Expression_3$.Type))
  CG.Comparison($Expression_2$.Type.higherThan($Expression_3$.Type),
                $Expression_1$.Operator)

VALUE[FieldAccess: $Expression_1$ → $Expression_2$ $Expression_3$] () =
  ADDRESS[$Expression_2$] ()
  <LOAD> $Expression_1$.Type

VALUE[Indexing: $Expression_1$ → $Expression_2$ $Expression_3$] () =
  ADDRESS[$Expression_2$] ()
  <LOAD> $Expression_1$.Type

VALUE[IntLiteral: Expression → INT_CONSTANT] () =
  <PUSHI> INT_CONSTANT

VALUE[Invocation: $Expression_1$ → $Expression_2$ Expression*] () =
  For(Expression exp : Expression*){
      VALUE[exp] ()
  }
  CG.Call($Expression_2$.Name)

VALUE[Logical: Expression$_1$ → Expression$_2$ ('&&'|'||') Expression$_3$] () =
  VALUE[Expression$_2$] ()
  CG.ConvertTo(Expression$_2$.Type, IntType.getInstance())
  VALUE[Expression$_3$] ()
  CG.ConvertTo(Expression$_3$.Type, IntType.getInstance())
  CG.Logical(Expression$_1$.Operator)

VALUE[RealLiteral: Expression → REAL_CONSTANT] () =
  <PUSHF> REAL_CONSTANT

VALUE [Ternary: Expression$_1$ → Expression$_2$ Expression$_3$ Expression$_4$] () =
  int labelNumber = CG.getLabels(2)
  VALUE[Expression$_2$] ()
  <JZ> labelNumber
  VALUE[Expression$_3$] ()
  CG.ConvertTo(Expression$_3$.Type, Expression$_1$.Type)
  <JMP> labelNumber+1
  CG.Label(labelNumber)
  VALUE[Expression$_4$] ()
  CG.ConvertTo(Expression$_4$.Type, Expression$_1$.Type)
  CG.Label(labelNumber+1)

VALUE [UnaryMinus: Expression$_1$ → Expression$_2$] () =
  <PUSHI> 0
  CG.ConvertTo(IntType.getInstance(), Expression$_2$.Type)
  VALUE[Expression$_2$] ()
  <SUB> Expression$_2$.Type

VALUE [UnaryNot: Expression$_1$ → Expression$_2$] () =
  VALUE[Expression$_2$] ()
  <NOT>

VALUE [Variable: Expression → ID] () =
  ADDRESS[Expression] ()
  If (!Expression.Definition.Type.isReference)
    <LOAD> Expression.Type

3.3 Execute

EXECUTE [Program: Program → Definition*] () =
  For(Definition def : Definition*)
    Execute[def] ()
  <CALL MAIN>
  <HALT>

EXECUTE [Assignment: Expression$_1$ → Expression$_2$ Expression$_3$] () =
  ADDRESS [Expression$_2$] ()
  VALUE[Expression$_3$] ()
  CG.ConvertTo(Expression$_3$.Type, Expression$_2$.Type)
  <STORE> Expression$_3$.Type

EXECUTE [BreakInstruction: Statement] (lastLabel) =
  <JMP> lastLabel+1

```
EXECUTE [DefaultCase: Statement → Statement*] (currentLabel, lastLabel) =
   For (Statement st : Statement*){
        If(st instanceof BreakInstruction)
                EXECUTE [st] (lastLabel)
        Else
                EXECUTE [st] ()
   }
```

```
EXECUTE [DoWhileStatement: Statement → Statement* Expression] () =
   int labelNumber = CG.getLabels(1)
   CG.label(labelNumber)
   For (Statement st : Statement*)
        EXECUTE [st] ()
   VALUE [Expression] ()
   <JNZ> labelNumber
```

EXECUTE [ForStatement: Statement → $Expression_1$ $Expression_2$ $Expression_3$ Statement*] () =

```
   int labelNumber = CG.getLabels(2)
   EXECUTE [Expression1] ()
   CG.label(labelNumber)
   VALUE [Expression2] ()
   <JZ> labelNumber+1
   For (Statement st : Statement*)
        Execute [st] ()
   EXECUTE [Expression3] ()
   <JMP> labelNumber
   CG.label(labelNumber+1)
```

```
EXECUTE [FunctionDefinition: Definition → Type Statement*] () =
   <ENTER> Definition.bytesLocalVariables
   For (Statement st : Statement*){
         If (st instanceof Return)
                EXECUTE [st] (Definition)
         Else
                EXECUTE [st] ()
   }
   If (Definition.Type.ReturnType instanceof VoidType.getInstance())
         <RET> 0, Definition.bytesLocalVariables, Definition.bytesParameters
```

```
EXECUTE [IfStatement: Statement → Expression Statement1* Statement2*] () =
   int labelNumber = CG.getLabels(2)
   VALUE [Expression] ()
   <JZ> labelNumber
   For (Statement st : Statement1*)
         EXECUTE [st] ()
   <JMP> labelNumber+1
   CG.label(labelNumber)
   For (Statement st : Statement2*)
         EXECUTE [st] ()
   CG.label(labelNumber+1)
```

```
EXECUTE [Invocation: Expression1 → Expression2 Expression*] () =
   For (Expression exp : Expression*)
         VALUE [exp] ()
   CG.call(Expression2.Name)
   If (!(Expression1.Type instanceof VoidType))
         <POP> Expression1.Type
```

```
EXECUTE [NormalCase: Statement → Expression Statement*] (currentLabel, lastLabel)
=
   VALUE [Expression] ()
   <EQ> Expression.Type
   <JZ> currentLabel+1
   For (Statement st : Statement*){
         If(st instanceof BreakInstruction)
                EXECUTE[st] (lastLabel)
         Else
                EXECUTE[st] ()
   }


EXECUTE [Read: Statement → Expression*] () =
   For (Expression exp : Expression*){
         ADDRESS [exp] ()
         <IN> exp.Type
         <STORE> exp.Type
   }

EXECUTE [Return: Statement → Expression] (Definition) =
   VALUE[Expression] ()
   CG.ConvertTo(Expression.Type, Definition.Type.ReturnType)
   <RET> Definition.bytesReturnType, Definition.bytesLocalVariables,
            Definition.bytesParameters
```

```
EXECUTE [SwitchCase: Statement → Expression Statement*] () =
   int lastLabel = Statement*.size - 1
   int labelNumber = CG.getLabels(last)
   For (Statement st : Statement*){
         CG.label(labelNumber)
         VALUE [Expression] ()
         EXECUTE [st] (labelNumber, lastLabel)
         labelNumber++
   }
   CG.label(labelNumber)


EXECUTE [WhileStatement: Statement → Expression Statement*] () =
   int labelNumber = CG.getLabels(2)
   CG.label(labelNumber)
   VALUE [Expression] ()
   CG.convertTo(Expression.Type, IntType.getInstance())
   <JZ> labelNumber+1
   For (Statement st : Statement*)
         EXECUTE [st] ()
   <JMP> labelNumber
   CG.label(labelNumber+1)


EXECUTE [Write: Statement → Expression*] () =
   For (Expression exp : Expression*){
         VALUE [exp] ()
         <OUT> exp.Type
   }
```