Translational Semantics

Use an attribute grammar to translate Wren into an assembly language (the target language).

- Translation preserves semantics
- Target language easily understood
- Example of operational semantics

Wren

- Assume programs satistify syntax specification, both context-free and context-sensitive.
- Parse declarations following concrete syntax, but no code will be generated for them.

Target Language

Assembly language for a simple single accumulator (Acc) machine

Instructions

LOAD	<name> or <const>	Load accumulator from named location or load constant
STO	<name>	Store accumulator to named location
GET	<name>	Input value to named location
PUT	<name>	Output value from named location
ADD	<name> or <const>	Acc ← Acc + <operand>
SUB	<name> or <const>	Acc ← Acc - <operand>
MULT	<name> or <const>	Acc ← Acc * <operand>
DIV	<name> or <const>	Acc ← Acc / <operand>
AND	<name> or 0 or 1	Acc ← Acc and <operand>
OR	<name> or 0 or 1	Acc ← Acc or <operand>
NOT		Acc ← not Acc
J	<label>	Jump unconditional
JF	<label>	Jump on false (Acc=0)
LABEL		Labeled instruction (L2 LABEL)
TSTLT		Test if Acc **L**ess **T**han zero
TSTLE		Test if Acc **L**ess than or **E**qual zero
TSTNE		Test if Acc **N**ot **E**qual zero
TSTEQ		Test if Acc **EQ**ual zero
TSTGE		Test if Acc **G**reater than or **E**qual zero
TSTGT		Test if Acc **G**reater **T**han zero
NO-OP		No operation
HALT		Halt execution

Example

```
program consec is
      var n : integer;
   begin
      n := 1;
      while n*(n+1)*(n+2) <> 800*(n+n+1+n+2) do
         n := n+1
      end while;
      write n;  write n+1;  write n+2
   end
```

```
    LOAD   1          LOAD   800        ADD    1
    STO    N          STO    T2         STO    N
L1  LABEL             LOAD   N          J      L1
    LOAD   N          ADD    N      L2  LABEL
    STO    T1         ADD    1          LOAD   N
    LOAD   N          ADD    N          STO    T1
    ADD    1          ADD    2          PUT    T1
    STO    T2         STO    T3         LOAD   N
    LOAD   T1         LOAD   T2         ADD    1
    MULT   T2         MULT   T3         STO    T1
    STO    T1         STO    T2         PUT    T1
    LOAD   N          LOAD   T1         LOAD   N
    ADD    2          SUB    T2         ADD    2
    STO    T2         TSTNE             STO    T1
    LOAD   T1         JF     L2         PUT    T1
    MULT   T2         LOAD   N          HALT
    STO    T1
```

Main Problem: Managing labels and temporary variables

Attribute Grammar

Attribute	Value
Name	Sequences of letters or digits
Temp	Natural numbers
SynLabel	Natural numbers
InhLabel	Natural numbers
OpCode	ADD, SUB, MULT, DIV
TestCode	TSTLT, TSTLE, TSTNE, TSTEQ, TSTGE, TSTGT
Code	Sequence of instructions of the following forms:

(LOAD, Name)	(STO, Name)
(GET, Name)	(PUT, Name)
(OpCode, Name)	NOT
(AND, Name)	(OR, Name)
(J, Name)	(JF, Name)
TestCode	(Name, LABEL)
NO-OP	HALT

Attributes (Page 207)

Nonterminal	Inherited Attributes	Synthesized Attributes
<program>	—	*Code*
<block>	—	*Code*
<dec sequence>	—	—
<declaration>	—	—
<variable list>	—	—
<type>	—	—
<cmd sequence>	*Temp, InhLabel*	*Code, SynLabel*
<command>	*Temp, InhLabel*	*Code, SynLabel*
<expr>	*Temp*	*Code*
<integer expr>	*Temp*	*Code*
<term>	*Temp*	*Code*
<element>	*Temp*	*Code*
<weak op>	—	*OpCode*
<strong op>	—	*OpCode*

<bool expr>	*Temp*	*Code*
<bool term>	*Temp*	*Code*
<bool element>	*Temp*	*Code*
<comparison>	*Temp*	*Code*
<relation>	—	*TestCode*
<variable>	—	*Name*
<identifier>	—	*Name*
<numeral>	—	*Name*
<letter>	—	*Name*
<digit>	—	*Name*

Notes

Name attribute for <variable>, <identifier>, <numeral>, <letter>, and <digit> handled as in Chapter 3.

Labels (L1, L2, …) must be unique throughout target program.

Temporary names (T1, T2, …) may be reused.

Inherited attribute *Temp* contains the numeral used in the last temporary name.

Inherited attribute *InhLabel,* containing the numeral used in the last label, is passed down to all commands; it is used by commands that involve branching.

Synthesized attribute *SynLabel* passes up the numeral used in the last label in the current command.

Expressions

<left operand> <operator> <right operand>

Assume n is value of inherited attribute *Temp*.

Template

Code(<left operand>)
STO T<n+1> (if n = 0, this is T1)
Code(<right operand>)
STO T<n+2> (if n = 0, this is T2)
LOAD T<n+1>
OpCode T<n+2>

OpCode is determined by the <operator>.

<intg expr> ::=
<term>
Code(<intg expr>) ← *Code*(<term>)
Temp(<term>) ← *Temp*(<intg expr>)

<intg expr> ::=
    <intg expr>$_2$ <weak op> <term>
      Code(<intg expr>) ←
concat(*Code*(<intg expr>$_2$),
   [(STO, temporary(temp+1))],
   Code(<term>),
   [(STO, temporary(temp+2)),
    (LOAD, temporary(temp+1)),
    (*OpCode*(<weak op>), temporary(temp+2))])

      Temp(<intg expr>$_2$) ← *Temp*(<intg expr>)
      Temp(<term>) ← *Temp*(<intg expr>)+1

<weak op> ::= **+**
      Opcode(<weak op>) ← ADD

<weak op> ::= **−**
      Opcode(<weak op>) ← SUB

Optimization

If *Code*(<right operand>) has length one
(a <variable> or a <numeral>), use
            Code(<left operand>)
            OpCode variable or numeral

<intg expr> ::=
    <intg expr>$_2$ <weak op> <term>
      Code(<intg expr>) ←
          concat(*Code*(<intg expr>$_2$),
              optimize(*Code*(<term>),
                  Temp(<intg expr>),
                  OpCode(<weak op>)))
      Temp(<intg expr>$_2$) ← *Temp*(<intg expr>)
      Temp(<term>) ← *Temp*(<integer expr>)+1

optimize(code, temp, opcode) =
   if length(code) = 1
      then [(opcode, secondField(first(code)))]
      else
      concat([(STO, temporary(temp+1))],
            code,
            [(STO, temporary(temp+2)),
              (LOAD, temporary(temp+1)),
              (opcode, temporary(temp+2))])

<term> ::=
    <element>
      Code(<term>) ← *Code*(<element>)
      Temp(<element>) ← *Temp*(<term>)

<term> ::=
    <term>$_2$ <strong op> <element>
      Code(<term>) ←
          concat(*Code*(<term>$_2$),
              optimize(*Code*(<element>),
                  Temp(<term>),
                  OpCode(<strong op>)))
      Temp(<term>$_2$) ← *Temp*(<term>)
      Temp(<element>) ← *Temp*(<term>)+1

<strong op> ::= *
      Opcode(<strong op>) ← MULT
<strong op> ::= /
      Opcode(<strong op>) ← DIV

<element> ::= <numeral>
    Code(<element>) ←
            [(LOAD,*Name*(<numeral>)]

<element> ::= <variable>
    Code(<element>) ←
            [(LOAD,*Name*(<variable>)]

<element> ::= (<intg expr>)
    Code(<element>) ← *Code*(<intg expr>)
    Temp(<intg expr>) ← *Temp*(<element>)

Auxiliary Functions

temporary(integer) = concat('T',string(integer))

label(integer) = concat('L',string(integer))

string(n) = see page 213

Comparisons

<integer expr>$_1$ <relation> <integer expr>$_2$

Template

code for <integer expr>$_1$
STO T<n+1>
code for <integer expr>$_2$
STO T<n+2>
LOAD T<n+1>
SUB T<n+2>
test condition

Note that comparision is with zero
<intg expr>$_1$ **<** <intg expr>$_2$ is true iff
(<intg expr>$_1$ – <intg expr>$_2$) < 0 is true.

<comparison> ::=
<intg expr>$_1$ <relation> <intg expr>$_2$
Code(<comp>) ← concat(
Code(<intg expr>$_1$),
optimize(*Code*(<intg expr>$_2$),
Temp(<comparison>),SUB),
[*TestCode*(<relation>)])

Temp(<intg expr>$_1$) ← *Temp*(<comp>)
Temp(<intg expr>$_2$) ← *Temp*(<comp>)+1

<relation> ::= **<**
TestCode(<relation>) ← TSTLT

<relation> ::= **<=**
TestCode(<relation>) ← TSTLE

<relation> ::= **<>**
TestCode(<relation>) ← TSTNE

<relation> ::= **=**
TestCode(<relation>) ← TSTEQ

<relation> ::= **>=**
TestCode(<relation>) ← TSTGE

<relation> ::= **>**
TestCode(<relation>) ← TSTGT

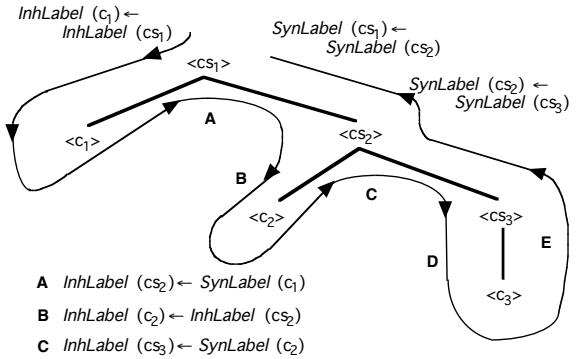
Commands

<program> ::= **program** <identifier> **is** <block>
Code(<program>) ←
concat(*Code*(<block>), [HALT])

<block> ::= <declaration sequence>
begin <command sequence> **end**
Code(<block>) ←
Code (<command sequence>)
Temp (<command sequence>) ← 0
InhLabel (<command sequence>) ← 0

<cmd seq> ::= <cmd>
Code(<cmd seq>) ← *Code*(<cmd>)
Temp(<cmd>) ← *Temp*(<cmd seq>)
InhLabel(<cmd>)← *InhLabel*(<cmd seq>)
SynLabel(<cmd seq>) ← *SynLabel*(<cmd>)

<cmd seq> ::= <cmd> ; <cmd seq>$_2$
Code(<cmd seq>) ←
concat(*Code*(<cmd>),*Code*(<cmd seq>$_2$))
Temp(<cmd>) ← *Temp*(<cmd seq>)
Temp(<cmd seq>$_2$) ← *Temp*(<cmd seq>)
InhLabel(<cmd>) ← *InhLabel*(<cmd seq>)
InhLabel(<cmd seq>$_2$) ← *SynLabel*(<cmd>)
SynLabel(<cmd seq>) ←
SynLabel(<cmd seq>$_2$)



Input Command

<command> ::= **read** <variable>

Code(<command>) ←
[(GET, *Name*(<variable>))]
SynLabel(<command>) ←
InhLabel(<command>)

Assignment Command

<command> ::= <variable> **:=** <expr>

Code(<command>) ←
concat(*Code*(<expr>),
[(STO,*Name*(<variable>))])
Temp(<expr>) ← *Temp*(<command>)
SynLabel(<command>) ←
InhLabel(<command>)

If Command

if <boolean expr> **then** <cmd seq> **end if**

Assume n = value of *InhLabel*

Template

Code(<boolean expr>)
JF L<n+1>
Code(<cmd seq>)
L<n+1> LABEL

<cmd> ::= **if** <boolean expr>
then <cmd seq> **end if**

Code(<cmd>) ←
concat(*Code*(<boolean expr>),
[(JF,label(*InhLabel*(<cmd>)+1))],
Code(<cmd sequence>),
[(label(*InhLabel*(<cmd>)+1),LABEL)])
Temp(<boolean expr>) ← *Temp*(<cmd>)
Temp(<cmd seq>) ← *Temp*(<cmd>)
InhLabel(<cmd seq>) ←
InhLabel(<cmd>)+1
SynLabel(<cmd>) ←
SynLabel(<cmd seq>)

While Command

while <boolean expr> **do**
<cmd seq> **end while**

Assume n = value of *InhLabel*

Template

L<n+1> LABEL
Code(<boolean expr>)
JF L<n+2>
Code(<cmd seq>)
J L<n+1>
L<n+2> LABEL

<cmd> ::=

while <boolean expr> **do**
<cmd seq> **end while**

Code(<cmd>) ← concat(
[(label(*InhLabel*(<cmd>)+1),LABEL)],
Code(<boolean expr>),
[(JF,label(*InhLabel*(<cmd>)+2))],
Code(<cmd seq>),
[(J,label(*InhLabel*(<cmd>)+1)),
(label(*InhLabel*(<cmd>)+2),LABEL)])
Temp(<boolean expr>) ← *Temp*(<cmd>)
Temp(<cmd seq>) ← *Temp*(<cmd>)
InhLabel(<cmd seq>) ←
InhLabel(<cmd>)+2
SynLabel(<cmd>) ←
SynLabel(<cmd seq>)

Implementing Code Generation

Use the same scanner as in Chapter 3.

Parser now produces a list of instructions representing the *Code* attribute for the program.

A “pretty print” predicate prints the assembly code program in tabular form.

```
program(Code) -->
        [program, ide(Ident), is],
        block(Code1),
        { concat(Code1,['HALT'],Code) }.

block(Code) -->
        decs,
        [begin],
        commandSeq(Code,0,0,SynLab),
        [end].
```


Command Sequence

```
cmdSeq(Code,Temp,InhLab,SynLab) -->
        cmd(Code1,Temp,InhLab,SynLab1),
        restcmds(Code2,Temp,SynLab1,SynLab),
        { concat(Code1,Code2,Code) }.

restcmds(Code,Temp,InhLab,SynLab) -->
        [semicolon],
        cmd(Code1,Temp,InhLab,SynLab1),
        restcmds(Code2,Temp,SynLab1,SynLab),
        { concat(Code1,Code2,Code) }.
restcmds([ ],Temp,InhLab,InhLab) --> [ ].
```

Commands

Input

```
cmd([['GET',Var]],Temp,Label,Label) -->
                                [read,ide(Var)].
```


Assignment

```
cmd(Code,Temp,Label,Label) -->
        [ide(Var), assign],
        expr(Code1,Temp),
        { concat(Code1,[['STO',Var]],Code) }.
```

If-Then Command

```
cmd(Code,Temp,InhLab,SynLab) -->
  [if], { InhLab1 is InhLab+1, label(InhLab1,Lab) },
  booleanExpr(Code1,Temp),
  [then], cmdSeq(Code2,Temp,InhLab1,SynLab),
  [end, if],
  { concat(Code1, [['JF',Lab]|Code2],
                        [[Lab,'LABEL']], Code) }.

label(Number,Label) :-  name('L', L1),
                        name(Number, L2),
                        concat(L1,L2,L),
                        name(Label,L).
```


If-Then-Else Command

```
cmd(Code,Temp,InhLab,SynLab) -->
  [if], { InhLab1 is InhLab+1,InhLab2 is InhLab+2,
          label(InhLab1,Lab1), label(InhLab2,Lab2) },
  booleanExpr(Code1,Temp), [then],
  cmdSeq(Code2,Temp,InhLab2,SynLab2),
  [else], cmdSeq(Code3,Temp,SynLab2,SynLab),
  [end, if]
  { concat(Code1, [['JF',Lab1]|Code2],
             [['J',Lab2], [Lab1,'LABEL']|Code3],
             [[Lab2,'LABEL']], Code) }.
```

While Command

```
cmd(Code,Temp,InhLab,SynLab) -->
  [while], { InhLab1 is InhLab+1, label(InhLab1,L1),
             InhLab2 is InhLab+2, label(InhLab2,L2) },
  booleanExpr(Code1,Temp), [do],
  cmdSeq(Code2,Temp,InhLab2,SynLab),
  [end,while],
  { concat([[L1,'LABEL']|Code1],[['JF',L2]|Code2],
                  [['J',L1],[L2,'LABEL']], Code) }.
```

Expressions

```
integerExpr(Code,Temp) -->
                term(Code1,Temp),
                restIntExpr(Code2,Temp)
                { concat(Code1,Code2,Code) }.

restIntExpr(Code,Temp) -->
        weakop(Op), { Temp1 is Temp+1 },
        term(Code1,Temp1),
        { optimize(Code1,OptCode1,Temp,Op) },
        restIntExpr(Code2,Temp)
        { concat(OptCode1,Code2,Code) }.

restIntExpr([ ], Temp) --> [ ].

weakop('ADD') --> [plus].
weakop('SUB') --> [minus].

term(Code,Temp) -->
        element(Code1,Temp),
        restTerm(Code2,Temp),
        { concat(Code1,Code2,Code) }.
```



```
restTerm(Code,Temp) -->
        strongop(Op), { Temp1 is Temp+1 },
        element(Code1,Temp1),
        { optimize(Code1,OptCode1,Temp,Op) },
        restTerm(Code2,Temp),
        { concat(OptCode1,Code2,Code) }.

restTerm([ ],Temp) --> [ ].

strongop('MULT') --> [times].
strongop('DIV') --> [divides].

element([['LOAD',Number]],Temp) -->
                                [num(Number)].

element([['LOAD',Var]],Temp) --> [ide(Var)].

element(Code,Temp) -->
    [lparen], integerExpr(Code,Temp), [rparen].
```


Optimization

```
optimize([['LOAD',Operand]],
            [[Opcode,Operand]],Temp,Opcode).


optimize(Code,OptCode,Temp,Op) :-
    Temp1 is Temp+1,
    Temp2 is Temp+2,
    temporary(Temp1,T1),
    temporary(Temp2,T2),
    concat([['STO',T1]|Code],
            [['STO',T2],['LOAD',T1],[Op,T2]],
            OptCode).
```


Comparisons

No optimization here.

```
comp(Code,Temp) -->
   { Temp1 is Temp+1, temporary(Temp1,T1),
     Temp2 is Temp+2, temporary(Temp2,T2) },
      integerExpr(Code1,Temp),
      testcode(Tst),
      integerExpr(Code2,Temp1),
      { concat(
          Code1,
          [['STO',T1]|Code2],
          [['STO',T2], ['LOAD',T1], ['SUB',T2], Tst],
          Code) }.

testcode('TSTEQ') --> [equal].
testcode('TSTNE') --> [neq].
testcode('TSTLT') --> [less].
testcode('TSTLE') --> [lteq].
testcode('TSTGT') --> [grtr].
testcode('TSTGE') --> [gteq].
```

Driver

```
go :- nl,
     write('>>> Translational Semantics <<<'), nl,
     write('Enter name of source file:  '), nl,
     getfilename(FileName), nl,
     see(FileName), scan(Tokens), seen,
     write('Scan successful'), nl, !,
     write(Tokens), nl, nl,
     program(Code,Tokens,[eop]), nl,
     write('Parse successful'), nl, nl,
     write(Code), nl, nl, prettyprint(Code), nl.
```

Try It

```
cp ~slonnegr/public/plf/translate .
```