CS4542 Compiler Design

WinZigC Compiler

Tarindu Jayatilaka - 160245K

Transduction Grammar

```
Winzig      -> 'program' Name ':' Consts Types Dcns
                 SubProgs Body Name '.'                  => "program";

Consts      -> 'const' Const list ',' ';'               => "consts"
            ->                                          => "consts";

Const       -> Name '=' ConstValue                      => "const";

ConstValue  -> '<integer>'
            -> '<char>'
            -> Name;

Types       -> 'type' (Type ';')+                       => "types"
            ->                                          => "types";

Type        -> Name '=' LitList                         => "type";

LitList     -> '(' Name list ',' ')'                    => "lit";

SubProgs    -> Fcn*                                     => "subprogs";

Fcn         -> 'function' Name '(' Params ')' ':' Name
                 ';' Consts Types Dclns Body Name ';'   => "fcn";

Params      -> Dcln list ';'                            => "params";

Dclns       -> 'var' (Dcln ';')+                        => "dclns"
            ->                                          => "dclns";

Dcln        -> Name list ',' ':' Name                   => "var";

Body        -> 'begin' Statement list ';' 'end'         => "block";

Statement   -> Assignment
            -> 'output' '(' OutExp list ',' ')'         => "output"
            -> 'if' Expression 'then' Statement
                 ('else' Statement)?                    => "if"
            -> 'while' Expression 'do' Statement        => "while"
            -> 'repeat' Statement list ';' 'until'
                 Expression                             => "repeat"
            -> 'for' '(' ForStat ';' ForExp ';'
                 ForStat ')' Statement                  => "for"
            -> 'loop' Statement list ';' 'pool'         => "loop"
            -> 'case' Expression 'of' Caseclauses
```

```
                  OtherwiseClause 'end'                    => "case"
            -> 'read' '(' Name list ',' ')'                => "read"
            -> 'exit'                                      => "exit"
            -> 'return' Expression                         => "return"
            -> Body
            ->                                             => "<null>";

OutExp      -> Expression                                  => "integer"
            -> StringNode                                  => "string";

StringNode  -> '<string>';

Caseclauses -> (Caseclause ';')+;

Caseclause  -> CaseExpression list ',' ':' Statement       => "case_clause";

CaseExpression
            -> ConstValue
            -> ConstValue '..' ConstValue                  => "..";

OtherwiseClause
            -> 'otherwise' Statement                       => "otherwise"
            -> ;

Assignment  -> Name ':=' Expression                        => "assign"
            -> Name ':=:' Name                             => "swap";

ForStat     -> Assignment
            ->                                             => "<null>";

ForExp      -> Expression
            ->                                             => "true";

Expression -> Term
            -> Term '<=' Term                              => "<="
            -> Term '<' Term                               => "<"
            -> Term '>=' Term                              => ">="
            -> Term '>' Term                               => ">"
            -> Term '=' Term                               => "="
            -> Term '<>' Term                              => "<>";

Term        -> Factor
            -> Term '+' Factor                             => "+"
            -> Term '-' Factor                             => "-"
            -> Term 'or' Factor                            => "or";

Factor      -> Factor '*' Primary                          => "*"
            -> Factor '/' Primary                          => "/"
            -> Factor 'and' Primary                        => "and"
            -> Factor 'mod' Primary                        => "mod"
            -> Primary;

Primary     -> '-' Primary                                 => "-"
```

```
            -> '+' Primary
            -> 'not' Primary                                => "not"
            -> 'eof'                                        => "eof"
            -> Name
            -> '<integer>'
            -> '<char>'
            -> Name '(' Expression list ',' ')'             => "call"
            -> '(' Expression ')'
            -> 'succ' '(' Expression ')'                    => "succ"
            -> 'pred' '(' Expression ')'                    => "pred"
            -> 'chr' '(' Expression ')'                     => "chr"
            -> 'ord' '(' Expression ')'                     => "ord";

Name        -> '<identifier>';
```

AST Grammar

```
Winzig      -> <'program' Name Consts Types Dclns SubProgs Body Name>

Consts      -> <'consts' Const+>
            -> 'consts'

Const       -> <’const’ Name ConstValue>

ConstValue  -> '<integer>'
            -> '<char>'
            -> Name

Types       -> <'types' Type+>
            -> 'types'

Type        -> <’type’ Name LitList>

LitList     -> <'lit' Name+>

SubProgs    -> <'subprogs' Fcn*>

Fcn         -> <'fcn' Name Params Name Consts Types Dclns Body Name>

Params      -> <'params' Dcln+>

Dclns       -> <'dclns' Dcln+>
            -> 'dclns'

Dcln        -> <'var' Name+ Name>

Body        -> <'block' Statement+>

Statement   -> Assignment
            -> <'output' OutExp+>
            -> <'if' Expression Statement Statement?>
            -> <'while' Expression Statement>
```

```
            -> <'repeat' Statement+ Expression>
            -> <'for' ForStat ForExp ForStat Statement>
            -> <'loop' Statement+>
            -> <'case' Expression Caseclause+ OtherwiseClause>
            -> <'read' Name+>
            -> 'exit'
            -> <'return' Expression>
            -> Body
            -> '<null>'

OutExp      -> <'integer' Expression>
            -> <'string' StringNode>

StringNode  -> '<string>'

Caseclause  -> <'case_clause' CaseExpression+ Statement>

CaseExpression
            -> ConstValue
            -> <'..' ConstValue ConstValue>

OtherwiseClause
            -> <'otherwise' Statement>
            ->

Assignment  -> <'assign' Name Expression>
            -> <'swap' Name Name>

ForStat     -> Assignment
            -> '<null>'

ForExp      -> Expression
            -> 'true'

Expression  -> Term
            -> <'<=' Term Term>
            -> <'<' Term Term>
            -> <'>=' Term Term>
            -> <'>' Term Term>
            -> <'=' Term Term>
            -> <'<>' Term Term>

Term        -> Factor
            -> <'+' Term Factor>
            -> <'-' Term Factor>
            -> <'or' Term Factor>

Factor      -> <'*' Factor Primary>
            -> <'/' Factor Primary>
            -> <'and' Factor Primary>
            -> <'mod' Factor Primary>
            -> Primary
```

```
Primary     -> <'-' Primary>
            -> Primary
            -> <'not' Primary>
            -> 'eof'
            -> Name
            -> '<integer>'
            -> '<char>'
            -> <'call' Name Expression+>
            -> Expression
            -> <'succ' Expression>
            -> <'pred' Expression>
            -> <'chr' Expression>
            -> <'ord' Expression>

Name        -> '<identifier>'
```

Simplified AST Grammar

```
P    -> <'program' '<identifier>' E E E E E '<identifier>'>

E    -> <'fcn' '<identifier>' E '<identifier>' E E E E '<identifier>'>
     -> <'var' '<identifier>'+ '<identifier>'>
     -> <'swap' '<identifier>' '<identifier>'>
     -> <’const’ '<identifier>' E>
     -> <’type’ '<identifier>' E>
     -> <'assign' '<identifier>' E>
     -> <'call' '<identifier>' E+>
     -> <'lit' '<identifier>'+>
     -> <'read' '<identifier>'+>
     -> <'subprogs' E*>
     -> <'consts' E*>
     -> <'types' E*>
     -> <'dclns' E*>
     -> <'params' E+>
     -> <'block' E+>
     -> <'output' E+>
     -> <'loop' E+>
     -> <'repeat' E+ E>
     -> <'case_clause' E+ E>
     -> <'case' E E+ E>
     -> <'for' E E E E>
     -> <'if' E E E>
     -> <'if' E E>
     -> <'while' E E>
     -> <'..' E E>
     -> <'<=' E E>
     -> <'<' E E>
     -> <'>=' E E>
     -> <'>' E E>
     -> <'=' E E>
     -> <'<>' E E>
     -> <'+' E E>
```

```
    -> <'-' E E>
    -> <'-' E>
    -> <'*' E E>
    -> <'/' E E>
    -> <'mod' E E>
    -> <'and' E E>
    -> <'or' E E>
    -> <'return' E>
    -> <'string' E>
    -> <'otherwise' E>
    -> <'not' E>
    -> <'succ' E>
    -> <'pred' E>
    -> <'chr' E>
    -> <'ord' E>
    -> 'exit'
    -> 'true'
    -> 'eof'
    -> <'integer' E>
    -> '<integer>'
    -> '<char>'
    -> '<identifier>'
    -> '<null>'
    -> '<string>'
    ->
```

Target Machine Instruction Set

```
PC := 1;
Next:
case Code[PC] of
      save n:     stack[n] := stack[top--]
      load n:     stack[++top] := stack[n]
      negate:     stack[top] := -stack[top]
      not:        stack[top] := not stack[top]
      add:        t := stack[top--]; stack[top] := stack[top] + t
      subtract:   t := stack[top--]; stack[top] := stack[top] - t
      mul:        t := stack[top--]; stack[top] := stack[top] * t
      div:        t := stack[top--]; stack[top] := stack[top] / t
      equal:      t := stack[top--]; stack[top] := stack[top] = t
      read:       stack[++top] := get(input)
      print:      put(stack[top--])
      lit n:      stack[++top] := n
      goto n:     PC := n; goto Next
      iffalse n:  if stack[top--] = 0 then PC:=n; goto Next fi
      iftrue n:   if stack[top--] = 1 then PC:=n; goto Next fi
      stop:       halt
end;
++PC;
goto Next;
```

Attribute Grammar

Data Structures

Declaration Table.
Functions:

- *enter(name,l)*
  - Binds "name" with stack location "l".
  - Returns "l".
- *lookup(name)*
  - Returns the location of "name" .
  - Returns 0 if "name" is not found.

Files.
Functions:

- *gen(file, arg1 , ..., argn)*
  - Writes a new line to "file".
  - The line contains arg1 , ..., argn.
  - Returns the new, modified file.
- *open*
  - Creates a new file.
- *close*
  - Closes a file.

Attributes

- code: File of code generated.
- next: Label of the next instruction on the code file.
- error: File of semantic errors.
- top: Current (predicted) size of run-time stack.
- type: Type of subtree. Used for type-checking.

Convention

- a ↑ is the synthesized attribute a.
- a ↓ is the inherited attribute a.

Synthesized and Inherited Attributes

S(program) = { code ↑, error ↑ }
I(program) = { }
S(*) = { code ↑, next ↑, error ↑, top ↑, type ↑ }
I(*) = { code ↓, next ↓, error ↓, top ↓ }

Defaults

```
no kids:
a ↑(ε) = a ↓(ε)

n kids:
a ↓(1) = a ↓(ε)
a ↓(i) = a ↑(i-1), for 1 < i <= n
a ↑(ε) = a ↑(n)
```

P -> <'program' '<identifier:x>' E E E E E '<identifier:y>'>

```
code  ↓(2)  = Open
next  ↓(2)  = 1
top   ↓(2)  = 0
code  ↑(ε)  = close (gen (code ↑(6), "stop"))
error ↑(ε)  = close (if x = y
                then error ↑(6)
                else gen (error ↑(6), "program names don't match"))
```

E -> <'var' '<identifier>'+ '<identifier>'>

defaults

E -> <'assign' '<identifier:x>' E>

```
code  ↑(ε)  = if lookup("x") = 0
                then enter("x",top ↑(2)); code ↑(2)
                else gen (code ↑(2), "save", lookup("x"))
next  ↑(ε)  = if lookup("x") = 0
                then next ↑(2)
                else next ↑(2) + 1
top  ↑(ε)   = if lookup ("x") = 0
                then top ↑(2)
                else top ↑(2) - 1
error ↑(ε)  = if type ↑(2) = "integer"
                then error ↑(2)
                else gen (error ↑(2),"Assignment type clash")
type  ↑(ε)  = "statement"
```

E -> <'consts' E*>

defaults

E -> <'types' E*>

defaults

E -> <'dclns' E*>

defaults

E -> <'subprogs' E*>

defaults

E -> <'block' E+>

defaults

E -> <'output' E+>

```
for 1 < i <= n,
code  ↓(i)  = gen (code ↑(i-1), "print")
next  ↓(i)  = next ↑(i-1) + 1
top   ↓(i)  = top ↑(i-1) - 1
type  ↓(i)  = "statement"
error ↓(i)  = if type ↑(i-1) = "integer" or "string"
              then error ↑(i-1)
              else gen (error ↑(i-1), "Illegal type for output")

code  ↑(ε)  = gen (code ↑(n), "print")
next  ↑(ε)  = next ↑(n) + 1
top   ↑(ε)  = top ↑(n) - 1
type  ↑(ε)  = "statement"
error ↑(ε)  = if type ↑(n) = "integer" or "string"
              then error ↑(n)
              else gen (error ↑(n), "Illegal type for output")
```

E -> <'if' E E E>

```
code  ↓(2)  = gen (code ↑(1),"iffalse", next ↑(2) + 1)
next  ↓(2)  = next ↑(1) + 1
top   ↓(2)  = top ↑(1) - 1
code  ↓(3)  = gen (code ↑(2), "goto", next ↑(3))
next  ↓(3)  = next ↑(2) + 1
error ↓(2)  = if type ↑(1) = "boolean"
              then error ↑(1)
              else gen (error ↑(1), "Illegal expression for if")
error ↓(3)  = if type ↑(2) = "statement"
              then error ↑(2)
              else gen (error ↑(2), "Statement required for if")
error ↑(ε)  = if type ↑(3) = "statement"
              then error ↑(3)
              else gen (error ↑(3), "Statement required for if")
```

E -> <'if' E E>

```
code  ↓(2)  = gen(code ↑(1), "iffalse", next ↑(2))
next  ↓(2)  = next ↑(1) + 1
top   ↓(2)  = top ↑(1) - 1
error ↓(2)  = if type ↑(1)="bool"
              then error ↑(1)
              else gen(error ↑(1), "non-bool exp")
type ↑(ε) = "statement"
```

E -> <'while' E E>

```
code  ↓(2)  = gen (code ↑(1), "iffalse", next ↑(2) + 1)
next  ↓(2)  = next ↑(1) + 1
top   ↓(2)  = top ↑(1) - 1
code  ↑(ε)  = gen (code ↑(2), "goto", next ↓(ε))
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "statement"
error ↓(2)  = if type ↑(1) = "boolean"
              then error ↑(1)
              else gen (error ↑(1),"Illegal expression in while")
error ↑(ε)  = if type ↑(2) = "statement"
              then error ↑(2)
              else gen (error ↑(2),"Statement required in while")
```

E -> <'=' E E>

```
code  ↑(ε)  = gen (code ↑(2), "equal")
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "boolean"
top   ↑(ε)  = top ↑(2) - 1
error ↑(ε)  = if type ↑(1) = type ↑(2)
              then error ↑(2)
              else gen (error ↑(2), "Type clash in equal comparison")
```

E -> <'+' E E>

```
code  ↑(ε)  = gen (code ↑(2), "add")
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "integer"
top   ↑(ε)  = top ↑(2) - 1
error ↑(ε)  = if type ↑(1) = type ↑(2) = "integer"
              then error ↑(2)
              else gen (error ↑(2), "Illegal type for plus")
```

E -> <'-' E E>

```
code  ↑(ε)  = gen (code ↑(2), "subtract")
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "integer"
top   ↑(ε)  = top ↑(2) - 1
error ↑(ε)  = if type ↑(1) = type ↑(2) = "integer"
              then error ↑(2)
              else gen (error ↑(2), "Illegal type for minus")
```

E -> <'-' E>

```
code  ↑(ε)  = gen (code ↑(1), "negate")
next  ↑(ε)  = next ↑(1) + 1
type  ↑(ε)  = "integer"
error ↑(ε)  = if type ↑(1) = "integer"
              then error ↑(1)
              else gen (error ↑(1), "Illegal type for minus"))
```

E -> <'*' E E>

```
code  ↑(ε)  = gen (code ↑(2), "mul")
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "integer"
top   ↑(ε)  = top ↑(2) - 1
error ↑(ε)  = if type ↑(1) = type ↑(2) = "integer"
              then error ↑(2)
              else gen (error ↑(2), "Illegal type for multiplication")
```

E -> <'/' E E>

```
code  ↑(ε)  = gen (code ↑(2), "div")
next  ↑(ε)  = next ↑(2) + 1
type  ↑(ε)  = "integer"
top   ↑(ε)  = top ↑(2) - 1
error ↑(ε)  = if type ↑(1) = type ↑(2) = "integer"
              then error ↑(2)
              else gen (error ↑(2), "Illegal type for multiplication")
```

E -> <'not' E>

```
code  ↑(ε)  = gen (code ↑(1), "not")
next  ↑(ε)  = next ↑(1) + 1
type  ↑(ε)  = "boolean"
error ↑(ε)  = if type ↑(1) = "boolean"
              then error ↑(1)
              else gen (error ↑(1),"Illegal type for not")
```

E -> <'integer' E>

defaults

E -> <'string' E>

defaults

E -> '<integer>'

```
code  ↑(ε)  = gen (code ↓(ε), "lit", "n")
next  ↑(ε)  = next ↓(ε) + 1
top   ↑(ε)  = top ↓(ε) + 1
type  ↑(ε)  = "integer"
```

E -> '<identifier>'

```
code  ↑(ε)  = gen (code ↓(ε), "load", lookup("x"))
next  ↑(ε)  = next ↓(ε) + 1
top   ↑(ε)  = top ↓(ε) + 1
type  ↑(ε)  = "integer"
error ↑(ε)  = if lookup("x") = 0
              then gen (error ↓(ε), "identifier un-initialized")
              else error ↓(ε)
```

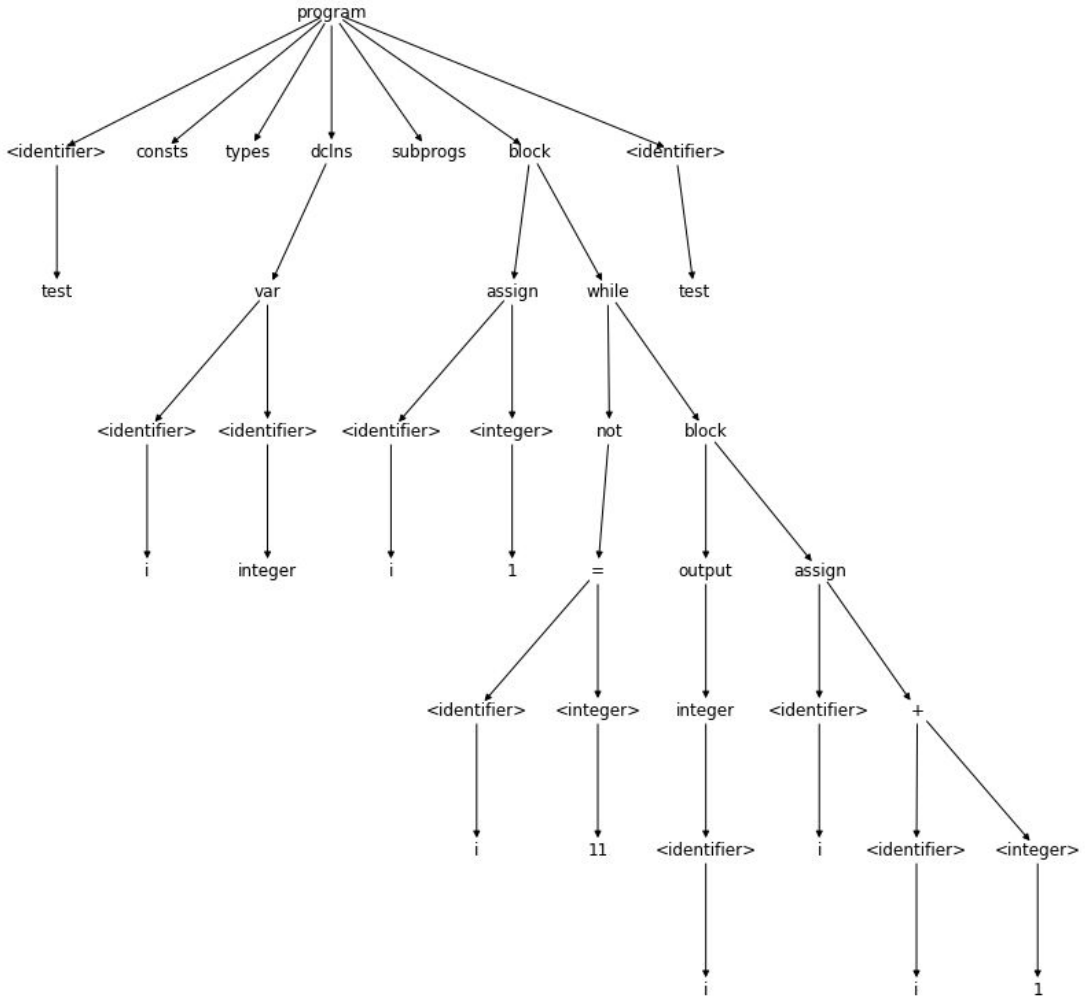
Sample Program

WinZigC Code

```
#this program prints numbers from 1 to 10

program test:
var i : integer;
begin
    i := 1;
    while not (i=11) do
    begin
        output (i);
        i := i + 1
        end
end test.
```

Abstract Syntax Tree



Corresponding Machine Code

```
1.   lit 1
2.   load 1
3.   lit 11
4.   equal
5.   not
6.   iffalse 14
7.   load 1
8.   print
9.   load 1
10.  lit 1
11.  add
12.  save 1
13.  goto 2
14.  stop
```