CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

http://www.cse.buffalo.edu/faculty/alphonce/SP17/CSE443/index.php
https://piazza.com/class/iybn4ndga1s3ei
shift/reduce conflict with easy fix

- pbblock and dbblock similarities seem to result in difficulties (grammar conflicts)

- Have all teams run into this?

- A simple fix seems to be: change the delimiters used in dblock from '(' and ')' to '[' and ']', as in:
  
  dblock is:
  
  '[' declaration-list ']'
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Three address code instructions (see 6.2.1, pages 364-5)

1. \( x = y \ op \ z \)
2. \( x = \ op \ y \)
3. \( x = y \)
4. \( \text{goto} \ L \)
5. \( \text{if} \ x \ \text{goto} \ L \ / \ \text{ifFalse} \ x \ \text{goto} \ L \)
6. \( \text{if} \ x \ \text{relop} \ y \ \text{goto} \ L \)
7. function calls: 
   - \( \text{param} \ x \)
   - \( \text{call} \ p, n \)
   - \( y = \text{call} \ p \)
   - return \( y \)
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = \&y, \ x = *y, *x = y \)

Need operators for various types: let’s use \(<t>\op, \text{as in} \)
\(i+ \) or \(r* \) or \(b<\)
dblocks (6.3.5 and 6.3.6)
records (in separate symbol table), sequence of declarations at start of sblock

definition → type identifier ':' dblock
{ st.put(identifier.lexeme, TYPE, dblock.type, dblock.width }  
dblock → '('
{ Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }  
declaration-list ')'
{ dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }

declaration-list → declaration ';' declaration-list

declaration-list → declaration

declaration → identifier ':'
{ id-list.type = identifier; } ← however you store types
identifier-list

identifier-list → identifier ( sBinOp constant ) ','
{ st.put(identifier.lexeme, VAR, identifier-list.type, offset);  
offset = offset + identifier-list.type.width; }

identifier-list

identifier-list → identifier ( sBinOp constant )
{ st.put(identifier.lexeme, VAR, identifier-list.type, offset);  
offset = offset + identifier-list.type.width; }

Just suggestions, not to be taken literally
dblocks (6.3.5 and 6.3.6)

records (in separate symbol table), sequence of declarations at start of sblock

definition → type identifier ':' dblock
    { st.put(identifier.lexeme, TYPE, dblock.type, dblock.width }  

dblock → '('
    { Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }
declaration-list ')
    { dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }

declaration-list → declaration ';&' declaration-list

declaration-list → declaration

declaration → identifier ':'
    { id-list.type = identifier; }  ← however you store types

identifier-list

identifier-list → identifier ( sBinOp constant ) ';
    { st.put(identifier.lexeme, VAR, identifier-list.type, offset);  
      offset = offset + identifier-list.type.width; }

identifier-list

identifier-list → identifier ( sBinOp constant )
    { st.put(identifier.lexeme, VAR, identifier-list.type, offset);  
      offset = offset + identifier-list.type.width; }

Just suggestions, not to be taken literally

We can specialize due to the structure of our grammar: see next slide!
**dblocks (6.3.5 and 6.3.6)**

Records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

\[
\text{dblock} \rightarrow '(', \quad \{ \quad \text{Env.push(st); st = new Env(); Stack.push(offset); offset = 0; } \quad \}
\]

\[
\text{declaration-list ')}' \quad \}
\]

\[
\text{dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); } \}
\]

---

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

\[
\text{integer: x} \quad \text{integer: y} \quad \text{integer: x} \\
\text{integer: y} \quad \text{real: x} \quad \text{real: z} \\
\text{integer: y} \quad \text{real: x} \quad \text{real: z} \\
\text{integer: y} \quad \text{real: x} \quad \text{real: z} \\
\text{integer: y} \quad \text{real: x} \quad \text{real: z}
\]
**dblocks (6.3.5 and 6.3.6)**

Records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```
dblock -> '('
    { Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }
    declaration-list ')
    { dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
```

---

AT RUNTIME

```
{ ( integer : x , y )
push offset = 8 onto stack
{ ( real : x , z ) ... ... }
pop offset = 8 from stack
push offset = 8 onto stack
{ ( Boolean : y , character : z )
  offset = 10... ... }
pop offset = 8 from stack
}
```
**dblocks (6.3.5 and 6.3.6)**

records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```plaintext
dblock → '('
{ Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }
declaration-list ')
{ dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
```

### AT RUNTIME

- **offset = 0**
- (integer : x , y)
- push offset = 8 onto stack
- offset = 8
- offset = 16
- offset = 24
- (real : x , z) .......
- pop offset = 8 from stack
- push offset = 8 onto stack
- offset = 8
- offset = 9
- offset = 10
- (Boolean : y ; character : z )
- ...
- pop offset = 8 from stack
```
dblocks (6.3.5 and 6.3.6)
records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

dblock → '('
{ Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }
declaration-list ')
{ dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
dblocks (6.3.5 and 6.3.6) records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

dblock → '('
{ Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }
declaration-list ')
{ dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
**dblocks (6.3.5 and 6.3.6)**

records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

\[
dblock \rightarrow '(\n  \{ \text{Env.push(st); st = new Env(); Stack.push(offset); offset = 0; }\}
  \text{declaration-list ')}
  \{ \text{dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }\}
\]

AT RUNTIME

```
{ offset = 0
offset = 4
offset = 8

{ (integer : x, y )
push offset = 8 onto stack

text offset = 8
offset = 16
offset = 24

{ (real : x, z )} ... ...
pop offset = 8 from stack
push offset = 8 onto stack

{ (integer : x)
offset = 8
offset = 9
offset = 10

{ (Boolean : y ; character : z )
offset = ...

} pop offset = 8 from stack
}
```
Dealing with alignment

\[
\{ \begin{array}{c}
[ \text{Boolean} : a \ ; \text{integer} : x \ ; \text{character} \ c; \text{real} : y ] \\
[ \text{character} : d \ ; \text{integer} : r \ , s ] \ldots \\
[ \text{Boolean} : f , g \ ; \text{real} : t \ ; \text{character} \ h ] \ldots \\
\end{array} \}
\]

Blocks are not aligned.
Dealing with alignment

{ [ Boolean : a ; integer : x ; character c ; real : y ]

{ [ character : d ; integer : r , s ] ... }

{ [ Boolean : f , g ; real : t ; character h ] ... }

Blocks are not aligned, but memory wasted to padding
Dealing with alignment

{ [ Boolean : a ; integer : x ; character c; real : y ]

{ [ character : d ; integer : r , s ] ... }

{ [ Boolean : f , g ; real : t ; character h ] ... }

}
Dealing with alignment

Block is aligned, padding needed before embedded scope block.
Expressions (6.4)

exp → ...

exp → assignable
  { exp.addr = assignable.addr;
    exp.code = exp₁.code; ... }

exp → '( exp₁ )'
  { exp.addr = exp₁.addr;
    exp.code = exp₁.code; ... }

exp → exp₁ binaryOperator exp₂
  { exp.addr = new Temp();
    exp.code = exp₁.code || exp₂.code ||
    gen(exp.addr '==' exp₁.addr '!=' exp₂.addr; ... }
Assignables (6.4)

assignable -> identifier

{ assignable.addr = ...;
  assignable.isFunCall = whether identifier has a function type - check symbol table }

assignable -> assignable₁ ablock
assignable₁ can be function call: we can return pointers to functions & arrays

{ assignable.addr = ...;
  temporary of return value, if function call, else address of array element
  assignable.isFunCall = whether identifier has a function type - check symbol table }

assignable -> assignable₁ recOp identifier {...}
assignable₁ can be function call so we can return pointers to records
'assignable' is poorly named

statement -> assignable sBinOp expression ';
{ if assignable.isFunCall then ...error... }
resolving overloaded operators

Fig 6.27, p. 390: after type checking and possibly coercions, then 6.5.3 choose correct operation to perform

\[ E \rightarrow E_1 + E_2 \]
\[
\{ \text{E.type} = \max(\text{E1.type}, \text{E2.type}); \\
a_1 = \text{widen}(\text{E1.addr}, \text{E1.type}, \text{E.type}); \\
a_2 = \text{widen}(\text{E2.addr}, \text{E2.type}, \text{E.type}); \\
\text{E.addr} = \text{new Temp}(); \\
\text{gen}(\text{E.addr} = ' + ' a_1 ' + ' a_2); \} \]
resolving overloaded operators

Fig 6.27, p. 390: after type checking and possibly coercions, then 6.5.3 choose correct operation to perform

```c
E -> E1 + E2
{ E.type = max(E1.type, E2.type);
a1 = widen(E1.addr, E1.type, E.type);
a2 = widen(E2.addr, E2.type, E.type);
E.addr = new Temp();
gen(E.addr '==' a1 '+' a2); }
```

Are we doing int addition or floating point addition? i+ vs. f+ ??
Skip 6.5.4-6.5.6
Control flow

- Booleans to control flow
- Booleans as values
Boolean expressions

- !X
- X & Y
- X | Y

We will do short-circuit evaluation

if (X | Y & Z) then { A } else { B } is translated as

```plaintext
if X goto LA
ifFalse Y goto LB
ifFalse Z goto LB

LA: A
goto END

LB: B

END: (next instruction)
```
Boolean expressions

A more concrete example:

if ( r < s | r = s & 0 < s) then { A } else { B } is translated as

if r < s goto LA
ifFalse r = s goto LB
ifFalse 0 < s goto LB

LA: A
goto END
LB: B
END: (next instruction)
Boolean expressions

- A more concrete example:

\[ \text{if } ( r < s \mid r = s \& \& 0 < s ) \text{ then } S1 \text{ else } S2 \text{ is translated as} \]

```plaintext
if r < s goto LA
ifFalse r = s goto LS1
ifFalse 0 < s goto LS2
LS1:   A
goto END
LS2:   B
END: (next instruction)
```

- CMP r s
- BLT LA

"The compare instruction compares two values and updates condition codes ... based on the difference obtained from subtracting s from r ... The Zero (Z) flag is set if the difference is zero. Otherwise it is reset (0). The Negative (N) flag is set if the difference is negative ... the Carry (C) flag is set if there is a carry out when the difference is computed. The Overflow (V) flag is set if the result of the difference overflows a 32-bit value." [Schindler, p. 46]
Flow-of-Control (6.3.3)

if ( B ) then S1 else S2

B.true = newlabel()
B.false = newlabel()
S.next = S1.next = S2.next
S.code = B.code ||
label(B.true) || S1.code ||
gen('goto' S.next) ||
label(B.false) || S2.code
Flow-of-Control (6.3.3)

\[ S \rightarrow \text{if ( } B \text{ ) then } S1 \]

\[ B.\text{true} = \text{newlabel()} \]

\[ B.\text{false} = S.\text{next} = S1.\text{next} \]

\[ S.\text{code} = B.\text{code} \lor \text{label}(B.\text{true}) \lor S1.\text{code} \]

\[ \text{ifTrue: goto LS1} \]

\[ \text{ifFalse: goto END} \]
Flow-of-Control (6.3.3)

while ( B ) then S1

begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S1.code = label(begin) || B.code || label(B.true) || S1.code || gen('goto' begin)
"When E appears in S -> while (E) S1, method jump is called at node E.n
[...] When E appears in S -> id = E;, method rvalue is called at node E.n" [p. 408]
Translation of: \( x = a < b \land \land c < d \)

```plaintext
ifFalse a < b goto L1
ifFalse c < d goto L1
t = true
goto L2
L1:  t = false
L2:  x = t
```