CSE 401: Introduction to Compiler Construction

Professor: Craig Chambers
TA: Markus Mock

Text: *Compilers: Principles, Techniques, and Tools*, Aho *et al.*

Goals:

- learn principles & practice of language implementation
  - brings together theory & pragmatics of previous courses
- study interactions among:
  - language features
  - implementation efficiency
  - compiler complexity
  - architectural features
- gain experience with object-oriented design & C++
- gain experience working on a team

Prerequisites:

- 326, 341, 378
- very helpful: 322

Course Outline

Front-end issues:

- lexical analysis (scanning): characters → tokens
- syntax analysis (parsing): tokens → abstract syntax trees
- semantic analysis (typechecking): annotate ASTs

Midterm

Back-end issues:

- run-time storage representations
- intermediate & target code generation: ASTs → asm code
- optimizations

Final

Project

Start with compiler for PL/0, written in C++

Add:

- comments
- arrays
- call-by-reference arguments
- results of procedures
- for loops
- break statements
- and more...

Completed in stages over the quarter

Strongly encourage working in a 2-person team on project

Grading based on:

- correctness
- clarity of design & implementation
- quality of test cases

Grading

Project: 40% total
Homework: 20% total
Midterm: 15%
Final: 25%

Homework & projects due at the **start of class**

3 free late days, per person

- thereafter, 25% off per calendar day late

An example compilation

Sample PL/0 program: `squares.0`

```
module main;
   var x:int, result:int;
   procedure square(n:int);
   begin
      result := n * n;
   end square;
begin
   x := input;
   while x <> 0 do
      square(x);
      output := result;
      x := input;
   end;
end main.
```

First step: lexical analysis

“Scanning”, “tokenizing”

Read in characters, clump into **tokens**

- strip out whitespace in the process

Specifying tokens: regular expressions

Example:

```
Ident    ::= Letter AlphaNum*
Integer  ::= Digit+
AlphaNum ::= Letter | Digit
Letter   ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit    ::= '0' | ... | '9'
```

Second step: syntax analysis

“Parsing”

Read in tokens, turn into a tree based on syntactic structure

Specifying syntax: context-free grammars

EBNF is a popular notation for CFG's

Example:

```
Stmt     ::= AsgnStmt | IfStmt | ...
AsgnStmt ::= LValue := Expr ;
LValue   ::= Id
IfStmt   ::= if Test then Stmt [else Stmt] ;
Test     ::= Expr = Expr | Expr < Expr | ...
Expr     ::= Term + Term | Term - Term | Term
Term     ::= Factor * Factor | ... | Factor
Factor   ::= - Factor | Id | Int | ( Expr )
```

EBNF specifies *concrete syntax* of language

Parser usually constructs tree representing *abstract syntax* of language

Third step: semantic analysis

"Name resolution and typechecking"

Given AST:

- figure out what declaration each name refers to
- perform static consistency checks

Key data structure: symbol table

- maps names to info about name derived from declaration

Semantic analysis steps:

1. Process each scope, top down
2. Process declarations in each scope into symbol table for scope
3. Process body of each scope in context of symbol table

Fourth step: storage layout

Given symbol tables,
determine how & where variables will be stored at run-time

What representation for each kind of data?

How much space does each variable require?

In what kind of memory should it be placed?

- static, global memory
- stack memory
- heap memory

Where in that kind of memory should it be placed?

- e.g. what stack offset

Fifth step: intermediate & target code generation

Given annotated AST & symbol tables,
produce target code

Often done as three steps:

- produce machine-independent low-level representation of program (intermediate representation)
- perform machine-independent optimizations of IR (optional)
- translate IR into machine-specific target instructions
  - instruction selection
  - register allocation

The bigger picture

Compilers are *translators*

Characterized by

- input language
- target language
- degree of “understanding”

Compilers vs. interpreters

Compilers implement languages by translation
Interpreters implement languages directly

Embody different trade-offs among:

- execution speed of program
- start-up overhead, turn-around time
- ease of implementation
- programming environment facilities
- conceptual difficulty

Engineering issues

Portability

- ideal: multiple front-ends & back-ends
  sharing intermediate language

Sequencing phases of compilation

- stream-based
- syntax-directed

Multiple passes?

Lexical Analysis / Scanning

Purpose: turn character stream (input program)
into **token stream**

- parser turns token stream into syntax tree

Token:
group of characters forming basic, atomic chunk of syntax

Whitespace:
characters between tokens that are ignored

Separate lexical and syntax analysis

Separation of concerns / good design

- scanner:
  - handle grouping chars into tokens
  - ignoring whitespace
  - handling I/O, machine dependencies
- parser:
  - handle grouping tokens into syntax trees

Restricted nature of scanning allows faster implementation

- scanning is time-consuming in many compilers

Language design issues

Most languages today are “free-form”

- layout doesn’t matter
- use whitespace to separate tokens, if needed

Alternatives:

- Fortran, Algol 68: whitespace ignored
- Haskell: use layout to imply grouping

Most languages today have “reserved words”

- can’t be used as identifiers

Alternative: PL/I: have “keywords”

- keywords treated specially only in certain contexts, otherwise just idents

Most languages separate scanning and parsing

Alternative: C++: ***type*** vs. ***ident***

- parser wants scanner to distinguish types from vars
- scanner doesn’t know how things declared

Lexemes, tokens, and patterns

Lexeme: group of characters that form a token

Token: class of lexemes that match a pattern

Pattern: description of string of characters

Token may have attributes, if more than one lexeme in token

Needs of parser determine how lexemes grouped into tokens

Regular expressions

Notation for specifying patterns of lexemes in a token

Regular expressions:

- powerful enough to do this
- simple enough to be implemented efficiently
- equivalent to finite state machines

Syntax of regular expressions

Defined inductively

- base cases:
  the empty string (ε)
  a symbol from the alphabet (x)
- inductive cases:
  sequence of two RE's: E_1E_2
  either of two RE's: $E_1 | E_2$
  Kleene closure (zero or more occurrences) of a RE: E^*

Notes:

- can use parens for grouping
- precedence: * highest, sequence, | lowest
- whitespace insignificant

Notational conveniences

E^+ means 1 or more occurrences of E

E^k means k occurrences of E

`[`E`]` means 0 or 1 occurrence of E (optional E)

`{`E`}` means E^*

not`(`x`)` means any character in the alphabet but x

not`(`E`)` means any string of characters in the alphabet but those matching E

$E_1 - E_2$ means any string matching E_1 except those matching E_2

Naming regular expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum    ::= letter | digit
```

EBNF-like notation for RE's

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed

Using regular expressions to specify tokens

Identifiers

```
ident       ::= letter (letter | digit)*
```

Integer constants

```
integer     ::= digit+
sign        ::= + | -
signed_int  ::= [sign] integer
```

Real number constants

```
real        ::= signed_int
                 [fraction] [exponent]
fraction    ::= . digit+
exponent    ::= (E|e) signed_int
```

String and character constants

```
string      ::= " char* "
character   ::= ' char '

char        ::= not("|'|\) | escape
escape      ::= \("|'|\|n|r|t|v|b|a)
```

Whitespace

```
whitespace  ::= <space> | <tab> | <newline>
                | comment
comment     ::= /* not(*/) */
```

Regular expressions for PL/0 lexical structure

```
Program  ::= (Token | White)*

Token    ::= Id | Integer | Keyword | Operator |
             Punct
Punct    ::= ; | : | . | , | ( | )
Keyword  ::= module | procedure | begin | end |
             const | var | if | then | while |
             do | input | output | odd | int
Operator ::= := | * | / | + | - |
             = | <> | <= | < | >= | >
Integer  ::= Digit+
Id       ::= Letter AlphaNum*
AlphaNum ::= Letter | Digit
Digit    ::= 0 | ... | 9
Letter   ::= a | ... | z | A | ... | Z

White    ::= <space> | <tab> | <newline>
```

Building scanners from RE patterns

Convert RE specification into
deterministic finite state machine

- by eye
- mechanically by way of
  nondeterministic finite state machine

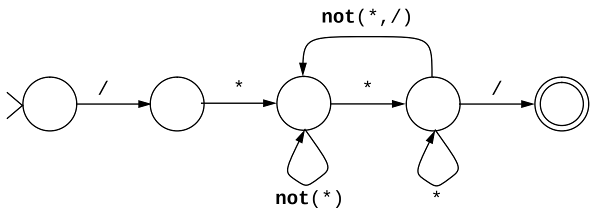
Convert DFA into scanner implementation

- by hand into collection of procedures
- mechanically into table-driven scanner

Finite State Machines/Automata

An FSA has:

- a set of states
  - one marked the initial state
  - some marked final states
- a set of transitions from state to state
  - each transition labelled with a symbol from the alphabet or ε



Operate by reading symbols and taking transitions,
beginning with the start state

- if no transition with a matching label is found, reject

When done with input, accept if in final state, reject otherwise

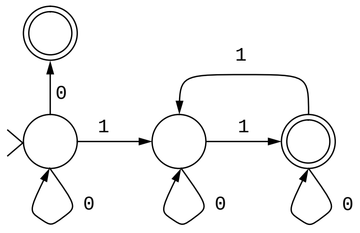
Determinism

FSA can be **deterministic** or **nondeterministic**

Deterministic: always know which way to go
- at most 1 arc leaving a state with particular symbol
- no ε arcs

Nondeterministic: may need to explore multiple paths

Example:



RE's map to NFA's easily
Can write code from DFA easily

Converting DFAs to code

Option 1: implement scanner using procedures
- one procedure for each token
- each procedure reads characters
- choices implemented using case statements

Pros
- straightforward to write by hand
- fast

Cons (if written by hand)
- more work than using a tool
- may have subtle differences from spec

Option 2: implement table-driven scanner
- rows: states of DFA
- columns: input characters
- entries: action
  - go to new state
  - accept token, go to start state
  - error

Pros
- convenient for automatic generation (e.g. `lex`)

Cons
- table lookups slower than direct code
- awkward to build by hand

Automatic construction of scanners

Approach:
1) Convert RE into NFA
2) Convert NFA into DFA
3) Convert DFA into table-driven code

RE ⇒ NFA

Define by cases

ε

x

$E_1\ E_2$

$E_1 \mid E_2$

E^*

NFA ⇒ DFA

Problem: NFA can “choose” among alternative paths,
while DFA must have only one path

Solution: **subset construction** of DFA

- each state in DFA represents *set of states in NFA*, all that the NFA might be in during its traversal

Subset construction algorithm

Given NFA with states and transitions

- label all NFA states uniquely

Create start state of DFA

- label it with the set of NFA states that can be reached by ε transitions (i.e. without consuming any input)

Process the start state

To process a DFA state *S* with label *{s1,..,sN}*:
For each symbol *s* in the alphabet:

- compute the set *T* of NFA states reached from any of the NFA states *s1,..,sN* by a *s* transition followed by any number of ε transitions
- if *T* not empty:
  - if a DFA state has *T* as a label, add a transition labeled *s* from *S* to *T*
  - otherwise create a new DFA state labeled *T*, add a transition labeled *s* from *S* to *T*, and process *T*

A DFA state is final iff
at least one of the NFA states in its label is final

Syntax Analysis/Parsing

Purpose: stream of tokens ⇒ **abstract syntax tree** (AST)

AST:

- captures hierarchical structure of input program
- primary representation of program, for rest of compiler

Context-free grammars (CFG’s)

Syntax specified using CFG’s

- RE’s not powerful enough
- context-sensitive grammars (CSG’s), general grammars (GG’s) too powerful

CFG’s: convenient compromise

- capture important structural characteristics
- some properties checked later during semantic analysis

Notation for CFG’s:
Extended Backus Normal (Naur) Form (EBNF)

EBNF description of PL/0 syntax

```
Program    ::=  module Id ; Block Id .
Block      ::=  DeclList begin StmtList end
DeclList   ::=  { Decl ; }
Decl       ::=  ConstDecl | ProcDecl | VarDecl
ConstDecl  ::=  const ConstDeclItem { , ConstDeclItem }
ConstDeclItem::=Id : Type = ConstExpr
ConstExpr  ::=  Id | Integer
VarDecl    ::=  var VarDeclItem { , VarDeclItem }
VarDeclItem::=  Id : Type
ProcDecl   ::=  procedure Id ( [ FormalDecl {, FormalDecl }
                ] ) ; Block Id
FormalDecl ::=  Id : Type
Type       ::=  int
StmtList   ::=  { Stmt ; }
Stmt       ::=  CallStmt | AssignStmt | OutStmt | IfStmt |
                WhileStmt
CallStmt   ::=  Id ( [ Exprs ] )
AssignStmt ::=  LValue := Expr
LValue     ::=  Id
OutStmt    ::=  output := Expr
IfStmt     ::=  if Test then StmtList end
WhileStmt  ::=  while Test do StmtList end
Test       ::=  odd Sum | Sum Relop Sum
Relop      ::=  <= | <> | < | >= | > | =
Exprs      ::=  Expr { , Expr }
Expr       ::=  Sum
Sum        ::=  Term { (+ | -) Term }
Term       ::=  Factor { (* | /) Factor }
Factor     ::=  - Factor | LValue | Integer | input | ( Expr )
```

Transition Diagrams

“Railroad diagrams”

- another more graphical notation for CFG’s
- look like FSA’s, where arcs can be labelled with nonterminals as well as terminals

Derivations and Parse Trees

Derivation: sequence of expansion steps,
beginning with start symbol,
leading to a string of terminals

Parsing: inverse of derivation

- given target string of terminals (a.k.a. tokens), want to recover nonterminals representing structure

Can represent derivation as a **parse tree**

- concrete syntax tree
- abstract syntax tree (AST)

Example grammar

```
E  ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /
```

Ambiguity

Some grammars are **ambiguous**:

- multiple different parse trees with same final string

Structure of parse tree captures much of meaning of program;
ambiguity ⇒ multiple possible meanings for same program

Designing a grammar

Concerns:

- accuracy
- readability, clarity
- unambiguity
- limitations of CFG's
- ability to be parsed by particular parsing algorithm
  - top-down parser ⇒ LL(k) grammar
  - bottom-up parser ⇒ LR(k) grammar

Famous ambiguities: "dangling else"

```
Stmt ::= ... |
         if Expr then Stmt |
         if Expr then Stmt else Stmt
```

"**if** e_1 **then if** e_2 **then** s_1 **else** s_2"

Resolving the ambiguity

Option 1: add a meta-rule
e.g. “`else` associates with closest previous `if`”

- works, keeps original grammar intact
- ad hoc and informal

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
                  if Expr then MatchedStmt
                          else MatchedStmt
UnmatchedStmt ::= if Expr then Stmt |
                  if Expr then MatchedStmt
                          else UnmatchedStmt
```

- formal, no additional rules beyond syntax
- sometimes obscures original grammar

Option 3: redesign the language to remove the ambiguity

```
Stmt ::= ... |
         if Expr then Stmt end |
         if Expr then Stmt else Stmt end
```

- formal, clear, elegant
- allows `StmtList` in `then` and `else` branch, no `begin`/`end` needed
- extra `end` required for every `if`

Another famous ambiguity: expressions

```
E  ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /
```

“a **+** b ***** c”

Resolving the ambiguity

Option 1: add some meta-rules,
e.g. precedence and associativity rules

Example:

```
E  ::= E Op E | - E | E ! | ( E ) | id
Op ::= + | - | * | / | ^ | = | < | and | or
```

operator	precedence	associativity
postfix `!`	highest	left
prefix `-`		right
`^`		right
`*`, `/`		left
`+`, `-`		left
`=`, `<`		none
`and`		left
`or`	lowest	left

Option 2: modify the grammar to explicitly resolve the ambiguity

Strategy:
- create a nonterminal for each precedence level
- expr is lowest precedence nonterminal,
  each nonterminal can be rewritten with higher precedence operator,
  highest precedence operator includes atomic exprs
- at each precedence level, use:
  - left recursion for left-associative operators
  - right recursion for right-associative operators
  - no recursion for non-associative operators

AST's

Abstract syntax trees represent only important aspects of concrete syntax trees
- no need for "signposts" like `( )`, `;`, `do`, `end`
- rest of compiler only cares about abstract structure
- can regenerate concrete syntax tree from AST when needed

AST extensions in project

Expressions:
- **true** and **false** constants
- array index expression
- fn call expression
- **and**, **or** operators
- tests are expressions
- constant expressions

Statements:
- **for** stmt
- **break** stmt
- **return** stmt
- **if** stmt with **else**
- array assignment stmt (similar to array index expr...)

Declarations:
- procedures with result type
- var parameters

Types:
- array type

Parsing algorithms

Given grammar, want to parse input programs

- check legality
- produce AST representing structure
- be efficient

Kinds of parsing algorithms:

- top-down
- bottom-up

Top-down parsing

Build parse tree for input program from the top (start symbol) down to leaves (terminals)

- leftmost derivation

Basic issue:

- when replacing a nonterminal with some r.h.s., how to pick which r.h.s.?

E.g.

```
Stmt    ::= Call | Assign | If | While
Call    ::= Id
Assign  ::= Id := Expr
If      ::= if Test then Stmts end |
            if Test then Stmts else Stmts end
While   ::= while Test do Stmts end
```

Solution: look at input tokens to help decide

Predictive parsing

Predictive parser:
top-down parser that can select correct rhs looking at at most *k* input tokens (the **lookahead**)

Efficient:

- no backtracking needed
- linear time to parse

Implementation of predictive parsers:

- table-driven parser
  - PDA: like table-driven FSA, plus stack to do recursive FSA calls
- recursive descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively

LL(*k*) grammars

Can construct predictive parser automatically/easily if grammar is **LL(*k*)**

- **L**eft-to-right scan of input, **L**eftmost derivation
- ***k*** tokens of lookahead needed, ≥ 1

Some restrictions:

- no ambiguity
- no **common prefixes** of length ≥ *k*:
  ```
  S ::= if Test then Ss end |
        if Test then Ss else Ss end | ...
  ```
- no **left recursion**:
  ```
  E ::= E Op E | ...
  ```
- a few others

Restrictions guarantee that, given *k* input tokens, can always select correct rhs to expand nonterminal

Eliminating common prefixes

Can **left factor** common prefixes to eliminate them

- create new nonterminal for common prefix and/or different suffixes

Before:

```
If      ::= if Test then Stmts end |
            if Test then Stmts else Stmts end
```

After:

```
If      ::= if Test then Stmts IfCont
IfCont  ::= end | else Stmts end
```

Grammar a bit uglier

Easy to do by hand in recursive-descent parser

Eliminating left recursion

Can rewrite grammar to eliminate left recursion

Before:

```
E ::= E + T | T
T ::= T * F | F
F ::= id | ...
```

After:

```
E ::= T { + T }
T ::= F { * F }
F ::= id | ...
```

Perhaps more readable after

- algorithm in book produces unsugared & ugly output

Easy to implement in hand-written recursive descent, too

Table-driven predictive parser

Can automatically convert grammar into PREDICT table

PREDICT(*nonterminal*, *input-sym*) ⇒ *production*

- selects the right production to take given a nonterminal to expand and the next token of the input

E.g.

```
stmt  ::= if expr then stmt else stmt |
          while expr do stmt |
          begin stmts end
stmts ::= stmt ; stmts | ε
expr  ::= id
```

PREDICT

	if	then	else	while	do	begin	end	id	;
stmt	1			2		3			
stmts	1			1		1	2		
expr								1	

Constructing PREDICT table

Compute FIRST for each r.h.s.

- FIRST(RHS) =
  {all tokens that can appear first in a derivation of RHS}

Compute FOLLOW for each nonterminal

- FOLLOW(X) =
  {all tokens that can appear after a derivation of X}

FIRST and FOLLOW computed mutually recursively

	FIRST	FOLLOW
S ::= **if** E **then** S **else** S		
| **while** E **do** S		
| **begin** Ss **end**		
Ss ::= S **;** Ss		
| ε		
E ::= **id**		

FIRST ⇒ PREDICT

Another example

Sugared:

```
E ::= T { (+|-) T }
T ::= F { (*|/) F }
F ::= - F | id | ( E )
```

Unsugared:

```
E  ::= T E′
E′ ::= (+|-) T E′ | ε
T  ::= F T′
T′ ::= (*|/) F T′ | ε
F  ::= - F | id | ( E )
```

	FIRST	FOLLOW
`E  ::= T E′`		
`E′ ::= (+|-) T E′`		
`      | ε`		
`T  ::= F T′`		
`T′ ::= (*|/) F T′`		
`      | ε`		
`F  ::= - F`		
`      | id`		
`      | ( E )`		

PREDICT and LL(1)

If PREDICT table has at most one entry in each cell, then grammar is LL(1)

- always exactly one right choice ⇒ fast to parse and easy to implement
- LL(1) ⇒ each column labelled by 1 token

Can have multiple entries in each cell

- e.g. if common prefixes or left recursion or ambiguity
- can patch table by hand, if know “right” answer
- or use more powerful parsing techniques

Recursive descent parsers

Write subroutine for each non-terminal

- each subroutine first selects correct r.h.s. by peeking at input tokens
- then consume r.h.s.
  - if terminal symbol, verify that it’s next & then advance
  - if nonterminal, call corresponding subroutine
- construct & return AST representing r.h.s.

PL/0 parser is recursive descent

PL/0 scanner routines:

- `Token* Get();`
- `Token* Peek();`
- `Token* Read(SYMBOL expected_kind);`
- `bool CondRead(SYMBOL expected_kind);`

Example

```
Stmt    ::= Assign | If
Assign  ::= id := Expr ;
If      ::= if Expr then Stmt [else Stmt] end ;
```

```
Stmt* Parser::ParseStmt() {
   Token* t = scanner->Peek();
   switch (t->kind()) {
      case IDENT:
         return ParseAssignStmt();
      case IF:
         return ParseIfStmt();
      default:
         Plzero->syntaxError(“expecting a stmt”);
   }
}
```

Example, continued

```
If       ::= if Expr then Stmt [else Stmt] end ;
```

```
IfStmt* Parser::ParseIfStmt() {
   scanner->Read(IF);
   Expr* test = ParseExpr();
   scanner->Read(THEN);
   Stmt* then_stmt = ParseStmt();
   Stmt* else_stmt;
   if (scanner->CondRead(ELSE)) {
      else_stmt = ParseStmt();
   } else {
      else_stmt = NULL;
   }
   scanner->Read(SEMICOLON);
   return new IfStmt(test, then_stmt, else_stmt);
}
```

Example, continued

```
Sum      ::= Term { (+ | -) Term }
```

```
Expr* Parser::ParseSum() {
   Expr* expr = ParseTerm();
   for (;;) {
      Token* t = scanner->Peek();
      if (t->kind() == PLUS || t->kind() == MINUS) {
         scanner->Get();  // skip over operator
         Expr* arg = ParseTerm();
         expr = new BinOp(t->kind(), expr, arg);
      } else {
         break;
      }
   }
   return expr;
}
```

Yacc

`yacc`: “**y**et **a**nother **c**ompiler-**c**ompiler”

Input: grammar, possibly augmented with action code
Output: C functions to parse grammar and perform actions

LALR(1) parser generator
- practical bottom-up parser
- more powerful than LL(1)

`yacc++`, `bison`, `byacc` are modern updates of `yacc`

Yacc input grammar

Examples:

```
assignstmt: IDENT GETS expr
          ;
ifstmt: IF test THEN stmts END
      | IF test THEN stmts ELSE stmts END
      ;

expr: term
    | expr '+' term
    | expr '-' term
    ;

factor: '-' factor
      | IDENT
      | INTEGER
      | INPUT
      | '(' expr ')'
      ;
```

Yacc with actions

Examples:

```
assignstmt: IDENT GETS expr
            { $$ = new AssignStmt($1, $3); }
           ;
ifstmt: IF test THEN stmtlist END
            { $$ = new IfStmt($2, $4); }
      | IF test THEN stmts ELSE stmts END
            { $$ = new IfElseStmt($2, $4, $6); }
      ;

expr: term { $$ = $1; }
    | expr '+' term
            { $$ = new BinOp(PLUS, $1, $3); }
    | expr '-' term
            { $$ = new BinOp(MINUS, $1, $3); }
    ;

factor: '-' factor { $$ = new UnOp(MINUS, $2); }
      | IDENT { $$ = new VarRef($1); }
      | INTEGER { $$ = new IntLiteral($1); }
      | INPUT { $$ = new InputExpr; }
      | '(' expr ')' { $$ = $2; }
      ;
```

Error handling

How to handle syntax error?

Option 1: quit compilation
+ easy
- inconvenient for programmer

Option 2: error recovery
+ try to catch as many errors as possible on one compile
- avoid streams of spurious errors

Option 3: error correction
+ fix syntax errors as part of compilation
- hard!!

Panic mode error recovery

When find a syntax error, skip tokens until reach a “landmark”

- landmarks in PL/0: **end**, **;**, **)**, **then**, **do**, ...
- once a landmark is found, will have gotten back on track

In top-down parser, maintain set of landmark tokens as recursive descent proceeds

- landmarks selected from terminals later in production
- as parsing proceeds, set of landmarks will change, depending on the parsing context

Example of error recovery

Grammar augmented with landmark sets

program	::=	$\texttt{stmt}_{\{\texttt{EOF}\}}$
stmt	::=	**if** $\texttt{expr}_{\{\textbf{then}\}}$ **then** stmt |
		while $\texttt{expr}_{\{\textbf{do}\}}$ **do** stmt |
		begin $\texttt{stmts}_{\{\textbf{end}\}}$ **end**
stmts	::=	$\texttt{stmt}_{\{;\}}$ **;** stmts | ε
expr	::=	**id** | **(** $\texttt{expr}_{\{)\}}$ **)**

Sample input

```
if x then begin while ( ...
```

Semantic Analysis/Checking

Final part of analysis half of compilation
- lexical analysis
- syntactic analysis
- semantic analysis

Afterwards comes synthesis half of compilation

Purposes:
- perform final checking of legality of input program, "missed" by lexical and syntactic checking
  - type checking, `break` stmt in loop, ...
- "understand" program well enough to do synthesis
  - e.g. relate assignments to & references of particular variable

Symbol tables

Key data structure during semantic analysis, code generation

Stores info about names used in program
- declarations add entries to symbol table
- uses of name look up corresponding symbol table entry to do checking, understanding

Symbol table interface in PL/0

```
class SymTabScope {
 public:
   SymTabScope(SymTabScope* enclosingScope);

   void enter(SymTabEntry* newSymbol);

   SymTabEntry* lookup(char* name);
   SymTabEntry* lookup(char* name,
                       SymTabScope*& retScope);

   // space allocation routines:
   void allocateSpace();
   int allocateLocal();
   int allocateFormal();

   ...
};
```

Symbol table entries

```
class SymTabEntry {
 public:
   char* name();
   Type* type();

   virtual bool isConstant();
   virtual bool isVariable();
   virtual bool isFormal();
   virtual bool isProcedure();

   // space allocation routine:
   virtual void allocateSpace(SymTabScope* s);

   // constants only:
   virtual int value();
   // variables only:
   virtual int offset(SymTabScope* s);

   ...
};

class VarSTE   : public SymTabEntry { ... };
class FormalSTE: public VarSTE      { ... };
class ConstSTE : public SymTabEntry { ... };
class ProcSTE  : public SymTabEntry { ... };
```

Implementation strategies

Abstraction: mapping from names to info

How to implement a map?

Option 1: linked list of key/value pairs

- requires only keys that can be compared for equality
- enter time: O(1)
- lookup time: O(*n*) (*n* entries in table)
- space cost: O(*n*)

Option 2: sorted binary search tree

- requires keys that can be ordered
- enter time: O(log *n*) expected, O(*n*) worst case
- lookup time: O(log *n*) expected, O(*n*) worst case
- space cost: O(*n*)

Option 3: hash table

- requires keys that can be hashed well
- requires good guess of size of table (*k*)
- enter time: O(1) expected, O(*n*) worst case
- lookup time: O(1) expected, O(*n*) worst case
- space cost: O(*k* + *n*)

Summary:

- use hash tables for big mappings,
  binary tree or linked list for small mappings
- ideal: self-reorganizing data structure

Nested scopes

How to handle nested scopes?

```
procedure foo(x:int, y:int);
   var z:bool;
   const y:bool = true;
   procedure bar(x:array[5] of bool);
      var y:int;
   begin
      ...
      x[y] := z;
   end bar;
begin
   ...
   while z do
      var z:int, y:int;
      y := z * x;
   end;
   output := x + y;
end foo;
```

Nested scopes

Want references to use closest textually-enclosing declaration

- static/lexical scoping, block structure

Simple solution: keep stack (linked list) of scopes

- stack represents static nesting structure of program
- top of stack = most closely nested

Used in PL/0

- each `SymTabScope` points to enclosing `SymTabScope` (`_parent`)
- maintains “down links,” too (`_children`)

Nested scope operations

When enter new scope during semantic analysis/type-checking:

- create a new, empty scope
- push it on top of scope stack

When encounter declaration:

- add entry to scope on top of stack
- check for duplicates in that scope only
  (allow shadowing of name declared in enclosing scopes)

When encounter use:

- search scopes for declaration, beginning with top of stack
- can find name in any scope

When exit scope:

- pop top scope off stack

Types

Types are abstractions of values that share common properties

Type checking uses types to compute whether operations on values will be legal

Taxonomy of types

Basic/atomic types:

- `int`, `bool`, `char`, `real`, `string`, ...
- `enum`(*value*$_1$, ..., *value*$_n$)
- user-defined types: `Stack`, `SymTabScope`, ...

Type constructors:

- `ptr`(*type*)
- `array`(*index-range*, *element-type*)
- `record`(*name*$_1$:*type*$_1$, ..., *name*$_n$:*type*$_n$)
- `union`(*type*$_1$, ..., *type*$_n$) or
  type$_1$ + ... + *type*$_n$
- `function`(*arg-types*, *result-type*) or
  arg-type$_1$ $\times$ ... $\times$ *arg-type*$_n$ $\rightarrow$ *result-type*

Parameterized types:

- functions returning types
  - Array<T>
  - HashTable<Key,Value>

Type synonyms

- give alternative name to existing type

Representing types in PL/0

```
class Type {
   virtual bool same(Type* t);
   bool different(Type* t) { return !same(t); }
   ...
};

class IntegerType   : public Type {...};
class BooleanType   : public Type {...};
class ProcedureType : public Type {
   ...
   TypeArray* _formalTypes;
};

IntegerType* integerType;
BooleanType* booleanType;
```

Type checking terminology

Static vs. dynamic typing

- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

Strong vs. weak typing

- strong: guarantees no illegal operations performed
- weak: can't make guarantees

	static	dynamic
strong		
weak		

Caveats:

- hybrids are common
- mistaken usages are common
- "untyped," "typeless" could mean "dynamic" or "weak"

Bottom-up type checking

Traverse AST graph from leaves up

At each node:

- recursively typecheck subnodes (if any)
- check legality of current node, given types of subnodes
- compute & return result type of current node (if any)

Needs info from enclosing context, too

- need to know types of variables referenced
  ⇒ pass down symbol table during traversal
- legality of e.g. `break`, `return` statements
  ⇒ pass down whether in loop, result type of function

Type checking in PL/0

Overview:

```
Type* Expr::typecheck(SymTabScope* s);
void Stmt::typecheck(SymTabScope* s);
void Decl::typecheck(SymTabScope* s);

Type* LValue::typecheck_lvalue(SymTabScope* s);

int Expr::resolve_constant(SymTabScope* s);

Type* TypeAST::typecheck(SymTabScope* s);
```

Type checking expressions

```
Type* IntegerLiteral::typecheck(SymTabScope* s) {
   // return result type
   return integerType;
}

Type* VarRef::typecheck(SymTabScope* s) {
   SymTabEntry* ste = s->lookup(_ident);

   // check for errors
   if (ste == NULL) {
      Plzero->typeError("undeclared var");
   }
   if (! ste->isConstant() &&
       ! ste->isVariable()) {
      Plzero->typeError("not a var or const");
   }

   // return result type
   return ste->type();
}
```

```
Type* BinOp::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* left = _left->typecheck(s);
   Type* right = _right->typecheck(s);

   // check the types of the operands
   switch(_op) {
    case PLUS: case MINUS: ...
      if ( left->different(integerType) ||
          right->different(integerType)) {
         Plzero->typeError(“args not ints”);
      }
      break;
    case EQL: case NEQ:
      if (left->different(right)) {
         Plzero->typeError(“args not same type”);
      }
      break;
   }

   // return result type
   switch (_op) {
    case PLUS: case MINUS: case MUL: case DIVIDE:
      return integerType;
    case EQL: case NEQ: ...
      return booleanType;
   }
}
```

Type checking statements

```
void AssignStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* lhs = _lvalue->typecheck_lvalue(s);
   Type* rhs = _expr->typecheck(s);

   // check legality of subexpression types
   if (lhs->different(rhs)) {
      Plzero->typeError(“lhs and rhs types differ”);
   }
}

void IfStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* test = _test->typecheck(s);

   // check legality of subexpression types
   if (test->different(booleanType)) {
      Plzero->typeError(“test not a boolean”);
   }

   // check nested statements
   for (int i = 0; i < _then_stmts->length(); i++) {
      _then_stmts->fetch(i)->typecheck(s);
   }
}
```

```
void CallStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   TypeArray* argTypes = new TypeArray;
   for (int i = 0; i < _args->length(); i++) {
      Type* argType = _args->fetch(i)->typecheck(s);
      argTypes->add(argType);
   }
   ProcType* procType = new ProcType(argTypes);

   // check callee procedure
   SymTabEntry* ste = s->lookup(_ident);
   if (ste == NULL) {
      Plzero->typeError(“undeclared procedure”);
   }
   Type* procType2 = ste->type();

   // check compatibility of actuals and formals
   if (procType->different(procType2)) {
      Plzero->typeError(“wrong arg types”);
   }
}
```

Type checking declarations

```
void VarDecl::typecheck(SymTabScope* s) {
   for (int i = 0; i < _items->length(); i++) {
      _items->fetch(i)->typecheck(s);
   }
}

void VarDeclItem::typecheck(SymTabScope* s) {
   Type* t = _type->typecheck(s);
   VarSTE* entry = new VarSTE(_name, t);
   s->enter(entry);
}
```

```
void ConstDecl::typecheck(SymTabScope* s) {
   for (int i = 0; i < _items->length(); i++) {
      _items->fetch(i)->typecheck(s);
   }
}

void ConstDeclItem::typecheck(SymTabScope* s) {
   Type* t = _type->typecheck(s);

   // typecheck initializer
   Type* exprType = _expr->typecheck(s);
   if (t->different(exprType)) {
      Plzero->typeError("init of wrong type");
   }

   // evaluate initializer
   int value = _expr->resolve_constant(s);

   ConstSTE* entry = new ConstSTE(_name, t, value);
   s->enter(entry);
}
```

```
void ProcDecl::typecheck(SymTabScope* s) {
   // create scope for body of procedure
   SymTabScope* body_scope = new SymTabScope(s);

   // enter formals into nested scope,
   // and construct procedure's type
   TypeArray* formalTypes = new TypeArray;
   for (int i = 0; i < _formals->length(); i++) {
      FormalDecl* formal = _formals->fetch(i);
      Type* t = formal->typecheck(body_scope);
      formalTypes->add(t);
   }
   ProcType* procType = new ProcType(formalTypes);

   // add entry for procedure in enclosing scope
   ProcSTE* entry = new ProcSTE(_name, procType);
   s->enter(procSTE);

   // typecheck procedure body last
   _block->typecheck(body_scope);
}

void Block::typecheck(SymTabScope* s) {
   _decls->typecheck(_scope);
   _stmts->typecheck(_scope);
}
```

Typechecking records, classes, modules, ...

```
type R = record begin
   public x:int;
   public a:array[10] of bool;
   private m:char;
end record;

var r:R;

... r.x ...
```

Need:
- represent record type including fields of record
- represent public vs. private nature of fields
- name user-defined record types
- access fields of record values

An implementation

Represent record type using a symbol table for fields

```
class RecordType: public Type {
   ...
   SymTabScope* _fields;
};
```

Add `TypeSTE` symbol table entries for user-defined types like `R`

For public vs. private, add boolean flag to `SymTabEntry`'s

To typecheck `r.x`:
- typecheck `r`
- check it's a record
- lookup `x` in `r`'s symbol table
- check that it's public,
  or that current scope is nested in record/class/module
- extract & return type of `x`

Type equivalence

When is one type equal to another?

- implemented in PL/0 with `Type::same` function

"Obvious" for atomic types like `int`, `string`, user-defined types

What about type constructors like arrays?

```
var a1   :array[10] of int;
var a2,a3:array[10] of int;
var a4   :array[20] of int;
var a5   :array[10] of bool;
```

Structural vs. name equivalence

Structural equivalence:
two types are equal if they have same structure

- if atomic types, then obvious
- if type constructors:
  - same constructor
  - recursively, equivalent arguments to constructor
- implement with recursive implementation of `same`

Name equivalence:
two types are equal if they came from the same textual occurrence of type constructor

- implement with pointer equality of `Type` instances

Rules can be extended to allow type synonyms

Aside: implementing binary operations

Want to dispatch on **two** arguments, not just receiver

In Cecil, using **multi-methods**:

```
bool same(Type* t1, Type* t2) { return false; }
bool same(IntegerType* t1,
          IntegerType* t2) { return true; }
bool same(ProcType* t1, ProcType* t2) {
  return same(t1->args, t2->args); }
```

In C++, using **double dispatching**:

```
class Type {
  virtual bool sameAsInteger(IntegerType* t) {
    return false; }
  virtual bool sameAsProc(ProcType* t) {
    return false; }
};

class IntegerType: public Type {
  bool same(Type* t) {
    return t->sameAsInteger(this); }
  bool sameAsInteger(IntegerType* t) {
    return true; }
}

class ProcType: public Type {
  bool same(Type* t) {
    return t->sameAsProc(this); }
  bool sameAsProc(ProcType* t) {
    return args->same(t->args); }
};
```

Type conversions and coercions

In C, can **explicitly convert**
an object of type `float` to one of type `int`

- can represent as unary operator
- typecheck, codegen normally

In C, can **implicitly coerce**
an object of type `int` to one of type `float`

- system must insert unary conversion operators as part of type checking

Implementing a language

Given type-checked AST program representation:

- might want to run it
- might want to analyze program properties
- might want to display aspects of program on screen for user
- ...

To run program:

- can interpret AST directly
- can generate target program that is then run recursively

Tradeoffs:

- time till program can be executed (turnaround time)
- speed of executing program
- simplicity of implementation
- flexibility of implementation
- conceptual simplicity

Interpreters

Create data structures to represent run-time program state

- values manipulated by program
- activation record for each called procedure
  - environment to store local variable bindings
  - pointer to calling activation record (**dynamic link**)
  - pointer to lexically-enclosing activation record/environment (**static link**)

EVAL loop executing AST nodes

Pros and cons of interpretation

+ simple conceptually, easy to implement
+ fast turnaround time, good programming environments
+ easy to support fancy language features

- slow to execute
  - data structure for value vs. direct value
  - variable lookup vs. registers or direct access
  - EVAL overhead vs. direct machine instructions
  - no optimizations across AST nodes

Compilation

Divide interpreter run-time into two parts:

- compile-time
- run-time

Compile-time does preprocessing

- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs

Compile-time processing

Decide representation of run-time data values

Decide where data will be stored

- registers
- format of stack frames
- global memory
- format of in-memory data structures (e.g. records, arrays)

Generate machine code to do basic operations

- just like interpreting expression,
  except generate code that will evaluate it later

Do optimizations across instructions if desired

Compile time vs. run time

Compile time	Run time
Procedure	Activation record, Stack frame
Scope, symbol table	Environment, (Contents of stack frame)
Variable	Memory location, Register
Lexically-enclosing scope	Static link
Calling procedure	Dynamic link

Run-Time Storage Layout

Plan how & where to keep data at run-time

Representation of

- int, bool, ...
- arrays, records, ...
- procedures

Placement of

- global variables
- local variables
- formal parameters
- results

Data layout

Determined by type of data

- scalar data based on machine representation
- aggregates group these together

Integer: use hardware representation
(2,4, and/or 8 bytes of memory, maybe aligned)

Bool: e.g. 0 or 1, 1 byte or word

Char: 1-2 bytes or word

Pointer: use hardware representation
(2,4, or 8 bytes, maybe two words if segmented machine)

Layout of records

Concatenate layout of fields, respecting alignment restrictions

```
r: record
     b: bool;
     i: int;
     m: record
          b: bool;
          c: char;
        end;
     j: int;
  end;
```

Layout of arrays

Repeat layout of element type, respecting alignment

```
a: array [5] of record
                 i: int;
                 c: char;
               end;
```

Array length?

Multi-dimensional arrays

Recursively apply layout rule to subarray first

```
a: array [3] of array[5] of record
                               i: int;
                               c: char;
                             end;
```

Leads to **row-major** layout

Alternative: **column-major**

String representation

String ≈ array of chars
- can use array layout rule to layout strings

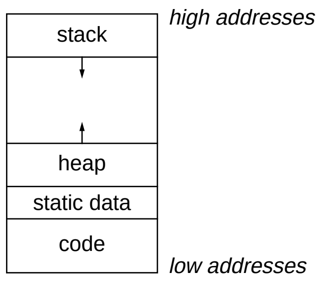
How to determine length of string at run-time?
- Pascal: strings have statically-determined length
- C: special terminating character
- High-level languages: explicit length field

Storage allocation strategies

Given layout of data structure,
where to allocate space for each variable/data structure?

Key issue: what is the **lifetime** (**dynamic extent**)
of a variable/data structure?
- whole execution of program (global variables)
  ⇒ **static** allocation
- execution of a procedure activation (formals, local vars)
  ⇒ **stack** allocation
- variable (dynamically-allocated data)
  ⇒ **heap** allocation

Parts of run-time memory (UNIX)



Code area
- read-only machine instruction area
- shared across processes running same program

Static data area
- place for read/write variables at fixed location in memory
- can be initialized, or cleared

Heap
- place for dynamically allocation/freed data
- can expand upwards through `sbrk` system call

Stack
- place for stack-allocated/freed data
- expands/contracts downwards automatically

Static allocation

Statically-allocate variables/data structures with global lifetime
- global variables
- compile-time constant strings, arrays, etc.
- `static` local variables in C, all locals in Fortran
- machine code

Compiler uses symbolic address
Linker determines exact address

Stack allocation

Stack-allocate variables/data structures with **LIFO** lifetime

- last-in first-out (stack discipline):
  data structure doesn't outlive previously allocated data structures on same stack

Procedure activation records usually allocated on a stack

- stack-allocated a.r. called a **stack frame**
- frame includes formals, locals, static link of procedure
- dynamic link = stack frame above

Fast to allocate & deallocate storage

Good memory locality

Problems with stack allocation

Stack allocation works only when can't have references to stack allocated data after data returns

Violated if first-class functions allowed

```
procedure foo(x:int):proctype(int):int;
   procedure bar(y:int):int;
   begin
      return x + y;
   end bar;
begin
   return bar;
end foo;

var f:proctype(int):int;
var g:proctype(int):int;

f := foo(3);
g := foo(4);

output := f(5);
output := g(6);
```

Violated if pointers to locals allowed

```
procedure foo(x:int):&int;
   var y:int;
begin
   y := x * 2;
   return &y;
end foo;

var z:&int;
var w:&int;

z := foo(3);
w := foo(4);

output := *z;
output := *w;
```

Heap allocation

Heap-allocate variables/data structures with unknown lifetime

- `new`/`malloc` to allocate space
- `delete`/`free`/garbage collection to deallocate space

Heap-allocate activation records (environments at least) of first-class functions

Relatively expensive to manage

Can have dangling references, storage leaks if don't free right

Stack frame layout

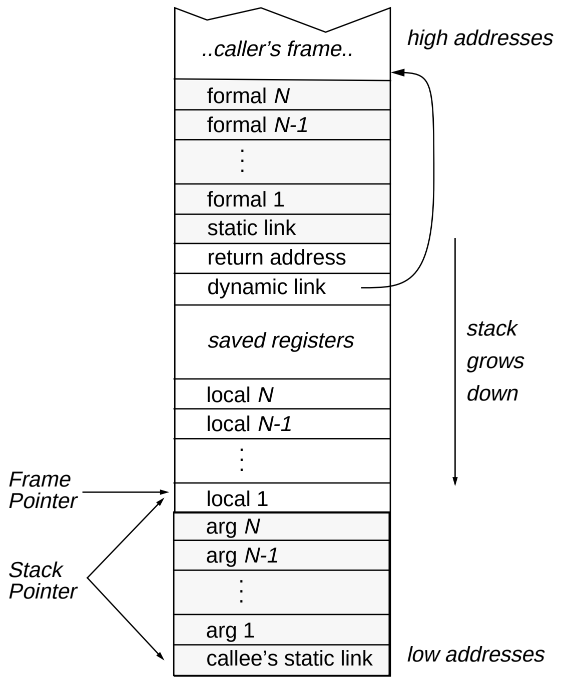
Need space for:

- formals
- local variables
- dynamic link (ptr to calling stack frame)
- static link (ptr to lexically-enclosing stack frame)
- other run-time data (e.g. return addr, saved registers)

Assign dedicated register(s) to support access to stack frames

- frame pointer (FP): ptr to beginning of stack frame (fixed)
- stack pointer (SP): ptr to end of stack (moves)

PL/0 stack frame layout



Calling conventions

Need to define responsibilities of caller and callee
in setting up, tearing down stack frame

Only caller can do some things
Only callee can do other things
Some things could be done by both

Need a protocol

PL/0 calling sequence

Caller:

- evaluates actual arguments, pushes them on stack
  - in what order?
  - alternative: 1st *k* arguments in registers
- pushes static link of callee on stack
  - or in register? before or after stack arguments?
- executes call instruction
  - return address stored in register by hardware

Callee:

- saves return address on stack
- saves caller's frame pointer (dynamic link) on stack
- saves any other registers needed by caller
- allocates space for locals, other data
  e.g. `sp := sp - size_of_locals - other_data`
  - locals stored in what order?
- sets up new frame pointer
  e.g. `fp := sp`
- starts running code...

PL/0 return sequence

Callee:

- deallocates space for locals, other data
  e.g. `sp := sp + size_of_locals + other_data`
- restores caller's frame pointer from stack
- restores return address from stack
- executes return instruction

Caller:

- deallocates space for callee's static link, args
  e.g. `sp := fp`
- continues execution...

PL/0 storage allocation

```
void SymTabScope::allocateSpace() {
   foreach sym
      sym->allocateSpace(s);
   foreach child scope
      child->allocateSpace();
}

void STE::allocateSpace(SymTabScope* s) {
   // do nothing
}
void VarSTE::allocateSpace(SymTabScope* s) {
   _offset = s->allocateLocal();
}
void FormalSTE::allocateSpace(SymTabScope* s) {
   _offset = s->allocateFormal();
}

int SymTabScope::allocateLocal() {
   int offset = _localsSize;
   _localsSize += sizeof(int);  // FIX THIS!
   return offset;
}
int SymTabScope::allocateFormal() { ... }
```

Static linkage

Need to connect stack frame to
stack frame holding values of lexically-enclosing variables

```
module M;
   var x:int;
   procedure P(y:int);
      procedure Q(y:int);
         begin R(x+y); end Q;
      procedure R(z:int);
         begin P(x+y+z); end R;
      begin Q(x+y); end P;
begin
   x := 1;
   P(2);
end M.
```

Accessing locals

If in same stack frame:

```
t := *(fp + local_offset)
```

If in lexically-enclosing stack frame:

```
t := *(fp + static_link_offset)
t := *(t + local_offset)
```

If farther away:

```
t := *(fp + static_link_offset)
t := *(t + static_link_offset)
...
t := *(t + static_link_offset)
t := *(t + local_offset)
```

Computing static link of callee is similar

At compile-time, need to calculate:

- difference in nesting depth of use and definition
- offset of local in defining stack frame

Parameter passing

When passing arguments, need to support right semantics

- lead to different representations for passed arguments and different code to access formals

Parameter passing semantics:

- call-by-value
- call-by-reference
- call-by-value-result
- call-by-result
- call-by-name
- call-by-need
- ...

Call-by-value

If formal is assigned, doesn't affect caller's value

```
var a:int;
procedure foo(x:int, y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement by passing copy of argument value

- trivial for ints, bools, etc.
- inefficient for arrays, records, strings, ...

Lisp, Smalltalk, ML, etc., use call-by-pointer-value

- "call-by-sharing"
- pass (copy of) pointer to data, pointer implicit

Call-by-reference

If formal is assigned, actual value is changed in caller

- change occurs immediately
- assumes actual is an lvalue

```
var a:int;
procedure foo(var x:int, var y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement by passing pointer to actual

- efficient for big data structures
- references to formal must do extra dereference!

If passing big immutable data (e.g. constant string) by value, can implement as call-by-reference

- can't assign to data, so can't tell it's a pointer

Call-by-value-result

If formal is assigned, final value copied back to caller when callee returns

- "copy-in, copy-out"

```
var a:int;
procedure foo(in out x:int, in out y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement as call-by-value, with assignment back when procedure returns

- more efficient for scalars than call-by-reference

Ada: `in out` either call-by-reference or call-by-value-result

- compiler can decide which is more efficient

Call-by-result

Formals assigned to return extra results;
no incoming actual value expected

- "out parameters"

```
var a:int;
procedure foo(out x:int, out y:int);
begin
   x := 1; y := a + 1;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement as in call-by-reference or call-by-value-result,
depending on desired semantics

Generating IR from AST's

Cases:

- expressions
- assignment statements
- control statements
- declarations ⇒ already done

Generating IR for expressions

How:

- tree walk, bottom up, left to right
- assign to a new temporary for each result

Pseudo-code:

```
Name IntegerLiteral::codegen(s) {
   result = new Name;
   emit(result := _value);
   return result;
}

Name BinOp::codegen(s) {
   Name e1 = _left->codegen(s);
   Name e2 = _right->codegen(s);
   result = new Name;
   emit(result := e1 _op e2);
   return result;
}
```

Example

```
module main;
   var z:int;
   procedure p(var q:int);
      var a: array[5] of array[10] of int;
      var b: int;
   begin
      b := 1 + 2;
      b := b + z;
      q := q + 1;
      b := a[4][8];
   end p;
begin
   z := 5;
   p(z);
end main.
```

IR for variable references

Two cases:

- if want l-value: load address
- if want r-value: load value @ address

To compute l-value:

```
Name VarRef::codegen_addr(s, int& offset) {
   ste = s->lookup(_ident, foundScope);
   if (!ste->isVariable()) ... // fatal error

   Name base = s->getFPOf(foundScope);
   offset = ste->offset();
   // base + offset = address of variable

   return base;
}
```

To compute r-value:

```
Name LValue::codegen(s) {
   int offset;
   Name base = codegen_addr(s, offset);
   Name dest = new Name;
   emit(dest := *(base+offset));
   return dest;
}
```

Shared by all lvalue syntax nodes (vars and arrays)

`VarRef::codegen` handles constants

IR for assignments

```
AssignStmt::codegen(s) {
   // compute address of l.h.s.:
   int offset;
   Name base = _lvalue->codegen_addr(s,
                                     offset);

   // compute value of r.h.s.:
   Name result = _expr->codegen(s);

   // do assignment:
   emit(*(base + offset) := result);
}
```

Accessing call-by-reference parameters

Formal parameter is **address** of actual, not value
⇒ need extra load

```
Reg VarRef::codegen_address(s, int& offset){
   ste = s->lookup(_ident, foundScope);
   // check for errors; defensive programming
   Name base = s->getFPOf(foundScope);
   offset = ste->offset();

   // NEW:
   if (ste->isFormalByRef()) {
      Name result = new Name;
      emit(result := *(base + offset);
      offset = 0;
   }

   return base;
}
```

Calling functions

New: by-ref arguments, return values

```
Name FunCall::codegen(s) {
  forall arguments, from left to right {
    if (arg is byValue) {
      // pass value of argument:
      name = arg->codegen(s);
      emit(push name);
    } else {
      // pass address of argument (NEW):
      int offset;
      base = arg->codegen_addr(s, offset);
      name = new Name;
      emit(name := base + offset);
      emit(push name);
    }
  }

  ...
```

```
  ...

  // compute & push static link:
  s->lookup(_ident, foundScope);
  Name link = s->getFPOf(foundScope);
  emit(push link);

  // generate call:
  emit(call _ident);

  // handle result (NEW):
  Name result = new Name;
  emit(result := RET0);
  return result;
}
```

IR for array accesses

Source code:

array_expr[*index_expr*]

Generated IR code:

```
a := <addr of array_expr>
i := <value of index_expr>
offset := i * <size of element type>
result := a + offset
// address of location = a + offset
```

Implementation of array access

```
Name ArrayRef::codegen_addr(s, int& offset){
  // compute address of array:
  Name base =
    _array->codegen_addr(s, offset);

  // compute value of index:
  Name i = _index->codegen(s);

  // scale index by elem size to get offset:
  int esize =
    _array_type->elem_type()->size();
  Name o = new Name;
  emit(o := i * esize);

  // compute final base address:
  Name result = new Name;
  emit(result := base + o);

  return result; // + offset!
}
```

Control structures

Rewrite control structures using
explicit labels and
conditional & unconditional branch IR instructions

E.g. `if` statement:

```
if test then stmts1 else stmts2 end;
     ⇒
  t1 := test
  if t1 = 0 goto _else  // conditional branch
  stmts1
  goto _done
_else:
  stmts2
_done:
```

Code for `if` codegen

```
void IfStmt::codegen(s) {
  // generate test expr into temp:
  Name t = _test->codegen(s);

  // generate conditional branch:
  Label else_lab = new Label;
  emit(if t = 0 goto else_lab);

  // generate then part:
  _then_stmts->codegen(s);

  // generate branch over else part:
  Label done_lab = new Label;
  emit(goto done_lab);

  // generate else part, with leading label:
  emit(else_lab:);
  _else_stmts->codegen(s);

  // finish up:
  emit(done_lab:);
}
```

`while` statement

```
while test do stmts end;
     ⇒
_loop:
  t1 := test
  if t1 = 0 goto _done
  stmts
  goto _loop
_done:
```

Short-circuiting

How to support short-circuit evaluation of `and` and `or`

Example:

```
if x <> 0 and y / x > 5 then
  b := y < x;
end;
```

Treat as control structure, not as operator:

```
  expr1 and expr2
     ⇒
  result := expr1
  if result = 0 goto _done
  result := expr2
_done:
```

IR codegen for break stmt

```
...
while ... do
   ...
   if ... then
      ...
      break;
   end;
   ...
end;
...
```

Target Code Generation

Input: intermediate language/representation

Intermediate languages:

- three-address code
- AST's + symbol table

Output: target language program

Target languages:

- absolute binary (machine) code
- relocatable binary code
- assembly code
- C

Task of code generator

Bridge the gap between intermediate code & target code

Intermediate code: machine independent
Target code: machine dependent

Instruction selection

- for each IR instruction (or sequence),
  select target language instruction (or sequence)

Register allocation

- for each IR variable,
  select target language register/stack location

Instruction selection

Given one or more IR instructions,
pick "best" sequence of target machine instructions
with same semantics

"best" = fastest, shortest

Difficulty depends on nature of target instruction set

- CISC: hard
  - lots of alternative instructions with similar semantics
  - lots of tradeoffs among speed, size
- RISC: easy
  - usually only one way to do something
  - closely resembles IR instructions
- C: easy, if C appropriate for desired semantics

Correctness a big issue, particularly if codegen complex

Example

IR code:

```
t3 := t1 + t2
```

Target code (MIPS):

```
add $3,$1,$2
```

Target code (SPARC):

```
add %1,%2,%3
```

Target code (68k):

```
mov.l d1,d3
add.l d2,d3
```

1 IR instruction can be several target instructions

Another example

IR code:

```
t1 := t1 + 1
```

Target code (MIPS):

```
add $1,$1,1
```

Target code (SPARC):

```
add %1,1,%1
```

Target code (68k):

```
add.l #1,d1
```

or

```
inc.l d1
```

Can have choices

- it's a pain to have choices; requires making decisions

Yet another example

IR code:

```
// push x onto stack
sp  := sp - 4
*sp := t1
```

Target code (MIPS):

```
sub $sp,$sp,4
sw $1,0($30)
```

Target code (SPARC):

```
sub %sp,4,%sp
st %1,[%sp]
```

Target code (68k):

```
mov.l d1,-(sp)
```

Several IR code instructions can combine to 1 target instruction
⇒ **hard!**

A final example

Source code:

```
a++; // "a" is a global variable
```

IR code:

Target code:

Instruction selection in PL/0

Do very simple instruction selection,
as part of generating code for AST node

- merged with ICG, because so simple

Interface to target machine: `assembler` class

- function for each kind of target instruction
- hides details of asm format, etc.

Register allocation

IR uses unlimited temporary variables

- makes ICG easy

Target machine has fixed resources for representing "locals"

- MIPS, SPARC: 31 registers,
  minus SP, FP, RetAddr, Arg1-4, ...
- 68k: 16 registers, divided into data and address regs
- x86: 4(?) general-purpose registers,
  plus several special-purpose registers

Registers *much* faster than memory

Must use registers in load/store RISC machines

Consequences:

- should/must try to keep values in registers if possible
- must free registers when no longer needed
- must be able to handle out-of-registers condition
  ⇒ **spill** some registers to home stack locations
- must interact with instruction selection, on CISCs
  ⇒ a real pain

Classes of registers

What registers can the allocator use?

Fixed/dedicated registers

- SP, FP, return address, ...
- claimed by machine architecture, calling convention, or
  internal convention for special purpose
- not easily available for storing locals

Scratch registers

- couple of registers kept around for temp values
  - e.g. loading a spilled value from memory in order to operate on it

Allocatable registers

- remaining registers free for register allocator to exploit

Classes of variables

What variables can the allocator try to put in registers?

Temporary variables: easy to allocate

- defined & used exactly once, during expression evaluation
  ⇒ allocator can free up register when used
- usually not too many in use at one time
  ⇒ less likely to run out of registers

Local variables: hard, but doable

- need to determine **last use** of variable in order to free reg
- can easily run out of registers ⇒
  need to make decision about which variables get regs
- what about assignments to local through pointer?
- what about debugger?

Global variables:
really hard, but doable as a research project

Simple register allocator design

Used in PL/0

Keep set of allocated registers as codegen proceeds
- `RegisterBank` class in PL/0

During codegen, allocate one from set
- `Reg temp = rb->getNew();`
- side-effects reg bank to record that `temp` is taken
- what if no registers available?

When done with register, release it
- `rb->free(temp);`
- side-effects reg bank to record that `temp` is free

Some codegen routines

```
Reg IntLiteral::codegen(Scope* s, RegBank* rb) {
   Reg dest = rb->getNew();
   TheAssembler->loadImmediate(dest, _value);
   return dest;
}

Reg BinOp::codegen(Scope* s, RegBank* rb) {
   Reg r1 = _left->codegen(s, rb);
   Reg r2 = _right->codegen(s, rb);
   rb->free(r1);
   rb->free(r2);
   Reg dest = rb->getNew();
   TheAsm->arith(_op, dest, r1, r2);
   return dest;
}

void AssignStmt::codegen(Scope* s, RegBank* rb) {
   Reg result = _expr->codegen(s, rb);
   int offset;
   Reg base = _lvalue->codegen_addr(s, rb, offset);
   TheAsm->store(result, base, offset);
   rb->free(base);
   rb->free(result);
}
```

An example

Source code:

```
var x;
...
x := x + 2 * (x - 1);
```

Function call codegen routine

```
Reg FnCall::codegen(Scope* s, RegBank* rb) {
   // eval & push arguments
   foreach arg, right to left {
      Reg a;
      if (pass by value) {
         a = arg->codegen(s, rb);
      } else {
         // pass by ref
         int offset;
         Reg base = arg->codegen_addr(s, rb, offset);
         Reg o = rb->getNew();
         TheAsm->loadImmediate(o, offset);
         rb->free(base);
         rb->free(o);
         a = rb->getNew();
         TheAsm->arith(PLUS, a, base, o);
      }
      TheAsm->push(SP, a);
      rb->free(a);
   }

   ...
   ...

   // eval & push static link
   Reg link = s->getFPOf(enclosingScope, rb);
   TheAsm->push(SP, link);
   rb->free(link);

   // save any allocated regs across call
   rb->saveRegs(s);

   // call
   TheAsm->call(_ident);

   // restore saved regs
   rb->restoreRegs(s);

   // pop off args & static link
   TheAsm->popMultiple(SP, (#args+1) * sizeof(int));

   // allocate temp reg for result of call
   Reg dest = rb->getNew();
   TheAsm->move(dest, RET0);

   // return result
   return dest;
}
```

Another example

Source code:

```
x := y + 4;
z := x * 8;
```

Optimizations

Identify inefficiencies in target or intermediate code

Replace with equivalent but better sequences

"Optimize" overly optimistic

- "usually improve" better

Scope of study for optimizations:

- **peephole**:
  look at adjacent instructions
- **local**:
  look at straight-line sequence of statements
- **global** (**int*ra*procedural**):
  look at whole procedure
- **int*er*procedural**:
  look across procedures

Larger scope ⇒ better optimization, more complexity

Peephole optimization

After code generation, look at adjacent instructions
(a "peephole" on the code stream)

- try to replace adjacent instructions with something faster

Example:

```
sw $8, 12($fp)
lw $12, 12($fp)
```

⇒

```
sw $8, 12($fp)
mv $12, $8
```

More examples

On 68k:

```
sub sp, 4, sp
mov r1, 0(sp)
   ⇒
mov r1, -(sp)

mov 12(fp), r1
add r1, 1, r1
mov r1, 12(fp)
   ⇒
inc 12(fp)
```

Do complex instruction selection through peephole optimization

Peephole optimization of jumps

Eliminate jumps to jumps

Eliminate jumps after conditional branches

“Adjacent” instructions = “adjacent in control flow”

Source code:

```
if a < b then
   if c < d then
      -- do nothing
   else
      stmt1;
   end;
else
   stmt2;
end;
```

Algebraic simplifications

“constant folding”, “strength reduction”

```
x := 3 + 4

x := x + 0
x := x * 1

x := x * 2
x := x * 8

x := x / 8

float x,y; x := (x + y) - y
```

Can be done by peephole optimizer, or by code generator

Local optimization

Analysis and optimizations within a **basic block**

Basic block: straight-line sequence of statements

- no control flow into or out of sequence

Better than peephole

Not too hard to implement

Machine-independent, if done on intermediate code

Local constant propagation

If variable assigned a constant,
  replace downstream uses of the variable with constant

Can enable more constant folding

Example:

```
const count:int = 10;
...
x := count * 5;
y := x ^ 3;
```

Unoptimized intermediate code:

```
t1 := 10
t2 := 5
t3 := t1 * t2
x := t3

t4 := x
t5 := 3
t6 := exp(t4, t5)
y := t6
```

Loca dead assignment elimination

Local common subexpression elimination

Avoid repeating the same calculation

Keep track of **available expressions**

Source:

```
... a[i] + b[i] ...
```

Unoptimized intermediate code:

```
t1 := i
t2 := t1 * 4
t3 := t2 + &a
t4 := *(fp + t3)

t5 := i
t6 := t5 * 4
t7 := t6 + &b
t8 := *(fp + t7)

t9 := t4 + t8
```

Int*ra*procedural (“global”) optimizations

Enlarge scope of analysis to whole procedure

- more opportunities for optimization
- have to deal with branches, merges, and loops

Can do constant propagation,
  common subexpression elimination, etc.
  at global level

Can do new things, e.g. **loop optimizations**

Optimizing compilers usually work at this level

Code motion

Goal: move **loop-invariant** calculations out of loops

Can do at source level or at intermediate code level

Source:

```
for i := 1 to 10 do
   a[i] := a[i] + b[j];
   z := z + 10000;
end;
```

Transformed source:

```
t1 := b[j];
t2 := 10000;
for i := 1 to 10 do
   a[i] := a[i] + t1;
   z := z + t2;
end;
```

Code motion at intermediate code level

Source:

```
for i := 1 to 10 do
   a[i] := b[j];
end;
```

Unoptimized intermediate code:

```
   *(fp + &i) := 1
_top:
   if *(fp + &i) > 10 goto _done
   t1 := *(fp + &j)
   t2 := t1 * 4
   t3 := fp + t2
   t4 := *(t3 + &b)
   t5 := *(fp + &i)
   t6 := t5 * 4
   t7 := fp + t6
   *(t7 + &a) := t4
   t9 := *(fp + &i)
   t10 := t7 + 1
   *(fp + &i) := t8
   goto _top
_done:
```

Loop induction variable elimination

For-loop index is **induction variable**

If used only to index arrays, can rewrite with pointers

Source:

```
for i := 1 to 10 do
   a[i] := a[i] + x;
end;
```

Transformed source:

```
for p := &a[1] to &a[10] do
   *p := *p + t;
end;
```

Global register allocation

Try to allocate local variables to registers

If two locals don’t overlap, then give to same register

Try to allocate most frequently-used variables to regs first

Example:

```
procedure foo(n:int, x:int):int;
   var sum:int, i:int;
begin
   sum := x;
   for i := 1 to n do
      sum := sum + i;
   end;
   return sum;
end foo;
```

How to do global optimizations?

Represent procedure by a **control flow graph**

Each **basic block** is a node in graph
Branches become edges in graph

Example:

```
procedure foo(n:int);
   var a:array[10] of int;
   var i:int, j:int;
begin
   j := 10;
   for i := 0 to 9 do
      a[i] := a[i] + (n - j * 2);
   end;
end foo;
```

Analysis of control flow graphs

To do optimization,
first analyze important info, then do transformations

Propagate info through graph
- At branches: copy info along both branches
- At merges: combine info, being conservative
- At loops: iterate to fixpoint

E.g. global constant propagation:
propagate *name*→*constant* mappings through graph

Int*er*procedural optimizations

Expand scope of analysis to procedures calling each other

Can do local, intraprocedural optimizations at larger scope

Can do new optimizations, e.g. **inlining**

Inlining

Replace procedure call with body of called procedure

Source:

```
const pi:real := 3.1415927;
proc circle_area(radius:int):int;
   begin
      return pi * (radius ^ 2);
   end circle_area;
...
r := 5;
...
output := circle_area(r);
```

After inlining:

```
const pi:real := 3.1415927;
...
r := 5;
...
output := pi * (r ^ 2);
```

Summary

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the global/intraprocedural level

Optimizations organized as collections of passes

Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write