引论

01 1946 **Ass** early 1950s, low dependent
Fortran John Backus hi sci-comp
Cobol busidat-proc **Lisp** symbolic comp
Lex, Syn, Sem, ICG|M-ICO, CG, M-DCO
Lex: breaks down the source code into a sequence of *lexemes* produce a *token*.
Syn: use token name -> IR -> grammar structure of token str, syntax tree
in-node: operation, child: arguments
describe the proper form of its programs
Sem: SynTr + symTab->SemConsistency
gather type-info for check, convert, ICG
describe the meaning of its programs
ICG: SynTr->TAC
MICO: faster, less mem&power, shorter
CG: IR->TarLang, alloc reg and mem
SymTab: front->back, vname, storage allocated, type, scope, proce name, arg num&type, pass by val/ref, return type
Compiler vs. Interpreter
1. hi-lvl->mac code on targetComputer
exec each stmt, no need into mac code
2. analyze stmt relation flows, optimize
less time to analyze, parse and exec
3. exec after successfully compiled
exec until 1st error met

词法

lexeme is a string of characters that is a lowest-level syntactic unit in programming languages
token is a syntactic category represent class of lexeme.<token name, attr value>
pattern: a description of the form that the lexemes of the token may take
string: finite sequence of symbols drawn from the **alphabet**(finite set of symbols)
lang: cntable set of str over fixed alpbet
closure * > concatenation > union |
regLang: lang can be defined by a regexp
NFA: A finite set of states S. A set of input symbols Σ, the input alphabet. The empty string $\varepsilon \notin \Sigma$. A transition function that gives a set of next states. A start state s0 from S. A set of accepting states F, subset of S.
DFA: no move on ε, unique edge<s,a>

语法

Term:Basic sym,str is formed from
NonT: Syntactic vars denote sets of str
Prod: Specify how the terms and nonT can be combined to form strings
head: NonT, **body**: 0 or more term/nonT
Derivation: StSym -> prod -> terms
sential form: $S \overset{*}{\Rightarrow} \alpha$
sentence: sentential form with no nonT
l/rmost drv: lrmost nonT to be replaced
parse tree: a graphical represent of a derivation that filters out the order in which productions are applied
root: StSym, leaf: terminal, int: NonT represents the applica of a production
m-1, drv->ParTr; 1-1 LRmost drv->ParTr
Ambiguity: >1 ParTr for some sentence
CFG>Regex, i->A_i, i-a-j -> $A_i \to aA_j$
{a^nb^n|n>0}, a^{k+1}, s >1 times, a^ib^j
$A \to Aa_1|Aa_n|b_1|b_m$ ->
$A \to b_1A'|b_mA'$, $A' \to a_1A'|a_nA'|\varepsilon$
Top-down: Construct a ParTr for the input str, start from the root and creating the nodes of the ParTr in preorder
pred: the prod applied for Lmost nonT?
match: terminals and chosen prod body
whether to hed->bdy by looking at next.
whether to head->ε by looking at next.
LL(1), $A \to a|b$:
1. FIRST(a) ∩ FIRST(b)=∅
2. If ε∈FIRST(b), FIRST(a)∩FOL(A)= ∅
No backtrack, **L**->r **L** most **1** lookahead.
A non-recursive predictive parser can be built by explicitly maintaining a stack
matched, stack, input, action
NULL, S$, input $, NOP
Bottom-up: Construct a ParTr for an input str begin at the leaves(terms) and up towards the root (Start Symbol)
finding a Rmost derivation (in reverse)
shift: Move an input sym onto the stack
reduce: Replace a str at the stktop with a nonT that can produce the str
No backtrack, **L**->r **R**-most **k** lookahead
Grammar available: LL < LR: virtually all
Cfgrations: represent the complete state of the parser (stack status + input status).
$(s_{0...m}, a_{i...n})$ $X_{0...m}a_{i...n}$ is a R-sentential
If there is no conflict during the ParTab construction, the grammar is SLR(1)
Cons: $\beta\alpha \to \beta A$, cannot follow by a
$[A \to \alpha\cdot, a]$ calls r $A \to \alpha$ only if the next symbol is a (a⊂FOL(A))
SLR(1) not enough,LR(1) too fine-grained
LALR(1): Keeps the lookahead symbols in the items; #states = SLR(1) ; Can deal with most common syntactic constructs of modern programming languages
Merge LR(1) items with same core(LR(0))
Merging not cause shift/reduce conflicts
$[A \to \alpha\cdot, a] \times [B \to \beta\cdot a\gamma, ?]$, !LR(1)
Merge may cause reduce/reduce conflicts
S'->S, S->aAd|bBd|aBe|bAe, A->c, B->c
{$[A \to c\cdot$,d],$[B \to c\cdot$,e]}, and swap d&e
merge: {$[A \to c\cdot$,d/e],$[B \to c\cdot$,d/e]}!
Merging states in LR(1) ParTab; If no r-r conflict, grammar is LALR(1), else not.
Grammars: CLR > LALR > SLR
#states: CLR > LALR = SLR
Driver: SLR = CLR = LALR

语法制导翻译

A **SDD** is a CFG with attributes and rules
synthesized: attrVal from child and self, evaluatd by single bot-up traverse ParTr
inherited: from self, parent and sibling
Not all nonT in a ParTr correspond to proper language constructs
Structure of ParTr may not match AST
S-attributed: every attr is synthesized
L-attributed: $A \to X_{1...n}$, $X_{1...j-1}$, A~X_j
AST: intern-construct, child-component
ParTr: intern-nonT, concrete SynTr
syntax-directed translation schemes: CFG with semantic actions(program frag) embedded within produXon bodies
Not all SDT's can be impl during parsing, introduc SemAct may cause inapplicable
Place a marker nonT to determine during

中间代码生成

TAC: x = y op z; if False x relop y goto L
param x; call p,n; y = call p,n; return y
x[i]=y;x=y[i];x=&y;x=*y;*x=y
$\leq$ 1 oprtrs, name/const/temp as oprnds
Quadruples: op,arg1,arg2,result
Triples: lineno, op, arg1, arg2
swap order optim make triples wrong
Indirect Tri: lineno(X)inst pointer(O)
Static single-assignment form: each name receives a single assignment$\phi(x_{12})$
TypExp: basic,Tname,array,record,→,×
NameEq: IFF same SynTr and labels
StructEq: IFF same TypExp, recur the Tr
A language is strongly typed if the compiler guarantees that the programs it accepts will run without type errors
Implicit automatic widening, exp narrow
base $+ i_1 * w_1 + i_2 * w_2$, w_1:row width
Basic idea of backpatching When jmp is generated, its target is tempoly left unspecified.Incomplete jmps grouped into lists. All jmps on a list have same target.Fill in the labels for incomplete jumps when the targets become known.

运行时环境

Code|Static|Heap|FreeMem|Stack
Static: decided by progtext, glbl const/var
Stack: local, actRec, during procCalls
Heap: outlive procCall creating, free/GC
ActivationTr: act as node, main as root
ProcCall: pre-order, return: post-order
ProcCalls and returns are managed by a run-time stack called the **control stack**
Activation Record
Actual params:Actual parameters used by the caller;Returned values:Values to

be returned to the caller;Ctrl link:Point to the actRec of the caller;Access link;Saved machine status:Information about the state of the machine before the call, including the return address and the contents of the registers used by the caller;Local data:Store the value of local variables;Temporaries:Temporary values such as those arising from the evaluation of expressions.

Calling sequences allocates an actRec on the stack and enters info into its fields. Pass args to the callee, transfer the ctrl to the first instruction of the callee;

1. caller evaluate actual parameters
2. caller store ra and sp into callee's AR
3. caller increment sp
4. callee save reg values and status info
5. callee init local data, exec

A **return sequence** pass the return values to the caller, transfer the ctrl to the caller so it can continue with the inst immediately after the ProcCall stmt

1. place return value nex to actual param
2. restores sp and other regs using info
3. goto return address set by caller

Alloc: Provide contiguous heap memory when a program requests memory for a var or obj. **Dealloc**: Return deallocated space to the pool of free space for reuse

Temporal locality: the memory locations accessed are likely to be accessed again within a short period of time.**Spatial**: memory locations close to the locations accessed are likely to be accessed within a short period of time

Frag: As the program alloc/dealloc memory, the heap is broken up into large number of small, noncontiguous holes.

1stfit, best fit, binnin,Doug Lea's Strategy

private bins: 16-24-32-...–512

larger-sz bin: 1024-..., 2048-...,

wilderness chunk: GBs, pages from OS

代码生成

static alloc:The size and layout of actRec are determined by the code generator via the information in the symbol table

stack alloc: using relative addresses for storage in actRec, sp to actRec on stktop

Loop: set of nodes, loop entry e, no out predecessor except e, node to e within L.

RegDescriptor: For each available reg, keeping track of the variable names whose current value is in that register

AddrDesc: For each program variable, keeping track of the locations where the current value of that var can be found

Register Allocation Algorithm

Assign specific values to certain registers

Simple design and impl but inefficient

Global Register Allocation

Assign registers to frequently used variables and keep these registers consistent across block boundaries, or estimate benifit static analysis/profiling

代码优化

Finding Local Common Subexpression

same oprt, same oprd(order), represent

Dead Code Elimination

delete any root having no live variables attached repeatedly.

The Use of Algebraic Identities

Eliminate computations: x+0=0,x/1=1

Reduction in strength: x/2=x*0.5

Constant folding: 2 × 3.14 = 6.28

The Data-Flow Analysis Schema

associate with every progpt a data-flow value that represents an abstraction of program states observed for that point

The data-flow problem is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s for all statements s

1. Constr based on the semantics of stmt
2. Constr based on the flow of control

-=========Computations=========-

$L \cup M = \{s | s \text{ is in } L \text{ or } s \text{ is in } M\}$

$LM = \{st | s \text{ is in } L \text{ and } t \text{ is in } M\}$

$L^* = \cup_{i=0}^{\infty} L^i, L^* = \cup_{i=1}^{\infty} L^i$

NFA/DFA construction is ez, omitted

FIRST

FIRST(X)={X};FIRST(X->ε),ε∈FIRST(X)

$X \to Y_1 ... Y_k$, prefix ε then a, all ε then ε

FIRST($X_1 ... X_n$), same as above

FOLLOW

Add $ to FOL(S), $\varepsilon \notin$ FOL

$A \to aBb$, FOL(B) += FIRST(b)

$A \to aB(\varepsilon)$, FOL(B) += FOL(A)

LL Parsing Table

$A \to a$, x in FIRST(a), M[A,x]+=$A \to a$

ε in FIR(a), b in FOL(A), M[A,b]+=$A \to a$

LR(0) Closure

$A \to a \cdot Bb$, $B \to c$, CLO(I) += $B \to \cdot c$

GOTO(I,X)

CLO({ $[A \to aX \cdot b]$ | $[A \to a \cdot Xb]$∈ I })

SLR Parsing Table

1. Canonical LR(0) collection for G'
2. $A \to \alpha \cdot a\beta$,Go$[I_i, a]$=$I_j$,ACT[i,a]=s j
   $A \to \alpha \cdot$,a∈FOL(A),act$[i, a]$=r $A \to \alpha$
   $S' \to S \cdot$ in I_i, ACT[i, $]=accept
3. GOTO$[I_i, A]$=I_j → GOTO[i,A]=j
4. Else error, init: CLO$[S' \to \cdot S]$

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	$\mathbf{id} * \mathbf{id}$ $	shift to 5
(2)	0 5	$ $\mathbf{id}$	$* \mathbf{id}$ $	reduce by $F \to \mathbf{id}$
(3)	0 3	$ F	$* \mathbf{id}$ $	reduce by $T \to F$
(4)	0 2	$ T	$* \mathbf{id}$ $	shift to 7
(5)	0 2 7	$ $T *$	$\mathbf{id}$ $	shift to 5
(6)	0 2 7 5	$ $T * \mathbf{id}$	$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$ $T * F$	$	reduce by $T \to T * F$
(8)	0 2	$ T	$	reduce by $E \to T$
(9)	0 1	$ E	$	accept

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR(1) CLOSURE

$[A \to \alpha \cdot B\beta, a]$,b∈FIR($\beta a$), $[B \to \cdot \gamma, b]$

Basic block partition

1. 1st inst is leader, target of jmp is leader, immediate follow jmp is leader
2. basic blk = [this leader,nex leder/EOF)

Flow graph

1. node: basic blk, edge: ed-st jmp/follow
2. entry/exit

Reaching definitions

A definition **d** of some variable **x** reaches a point **p** if there is a path from the progpt after **d** to **p,** such that **d** is not "killed" along that path

$f_{B(x)} = \text{gen}_B \cup (x - \text{kill}_B), \text{kill}_B = \cup\, \text{kill}_i$

$\text{gen}_B = \cup\, (\text{gen}_{n-i} - \text{kill}_{n-i+1...n})$

OUT[B] = gen$_B$ ∪ (IN[B] − kill$_B$)

IN[B] = ∪$_{\text{predecessor}}$OUT[P]