CS 131 Compilers: Discussion 8: Intermediate Representations: Basic Block and CFG & Mid Term Preparation2

杨易为 季杨彪 尤存翰

{yangyw,jiyb,youch}@shanghaitech.edu.cn

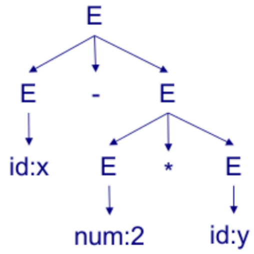
2021 年 4 月 24 日

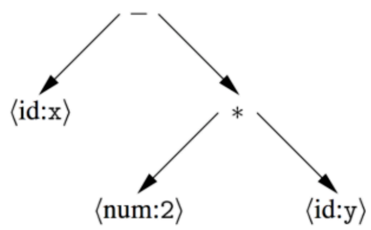
1 Intermediate Representations

An Intermediate Representation (IR) is an intermediate (neither source nor target) form of a program. There are various types of IRs:

1. Section

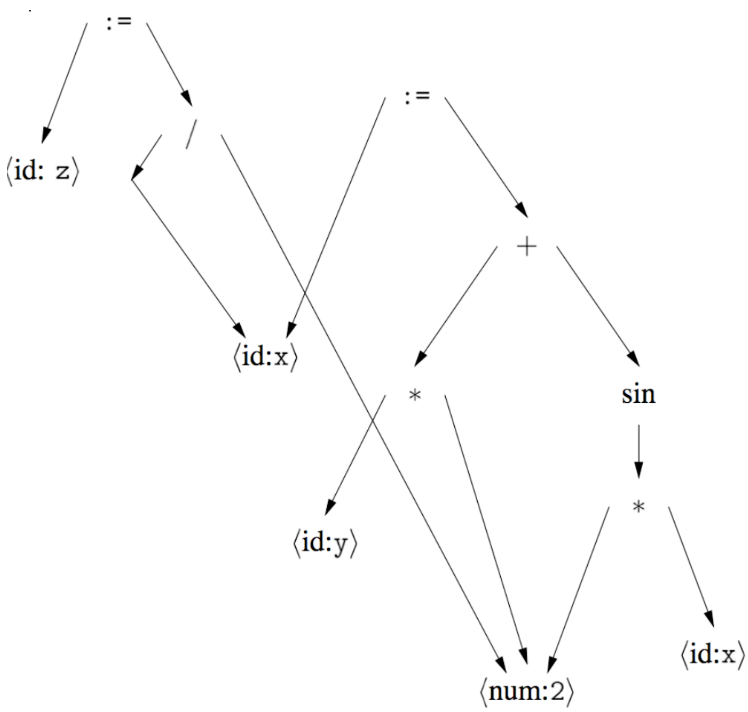
   (a) **Abstract Syntax Trees** (AST): a simplified Parse Tree.





   - + close to source compactdesc
   - + suitable for source-source translation
   - - Traversal & Transformations are expensive
   - - Pointer-intensive
   - - Memory-allocation-intensive

   (b) **Directed Acyclic Graphs** (DAG): DAG is an optimized AST, with identical nodes *shared*.

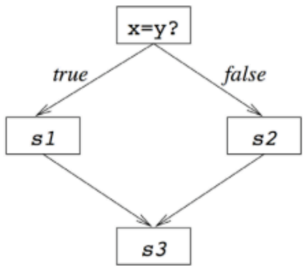


- + Explicit sharing
- + Exposes redundancy, more efficient, useful for dynamic pipelining analysis
- - Difficult to transform
- - Analysis usage Practical usage

(c) **Control Flow Graphs** (CFG): CFG is a flow chart of program execution. Is a conservative approximation of the Control Flow, because only one branch will be actually executed.
A Basic Block is a consecutive sequence of Statements $S_1, \ldots, S_n$, where flow must enter this block only at S_1, AND if S_1 is executed, then $S_2, \ldots, S_n$ are executed strictly in that order, unless one Statement causes halting.
- The Leader is the first Statement of a Basic Block
- A Maximal Basic Block is a maximal-length Basic Block

Nodes of a CFG are Maximal Basic Blocks, and Edges of a CFG represent control flows
- $\exists$ edge $b_1 \to b_2$ iff control may transfer from the last Statement of b_1 to the first Statement of b_2

```
if x = y then
   S1
else
   S2
end
S3
```



- + Most widely used form. Can cast static analysis on it.

(d) **Single Static Assignment** (SSA): SSA means every variable will only be assigned value ONCE (therefore single). Useful for various kinds of optimizations.

```
x := 3;
x := x + 1;
x := 7;
x := x*2;
```

➔

```
x1 := 3;
x2 := x1 + 1;
x3 := 7;
x4 := x3*2;
```

A ϕ -function generates an extra

assignment to "choose" from Branches or Loops. If Basic Block B has Predecessors $P_1, \ldots, P_n$, then $X = \phi(v_1, \ldots, v_n)$ assigns $X = v_j$ if control enters B from P_j.

1. 2-way Branch:

if (...)	if (...)
X = 5;	$X_0 = 5;$
else	else
X = 3;	$X_1 = 3;$
	$X_2 = \phi(X_0, X_1);$
Y = X;	$Y_0 = X_2;$

2. While Loop:

j = 1;	$j_5 = 1;$
S: // *while (j < x)*	S: $j_2 = \phi(j_5, j_4);$
if (j >= X)	if (j_2 >= X)
goto E;	goto E;
j = j+1;	$j_4 = j_2+1;$
goto S	goto S
E:	E:
N = j;	$N = j_2;$

i. ϕ is not an executable operation

ii. Number of ϕ arguments = Number of incoming edges

iii. Where to place a ϕ? If Basic Block B contains an assignment to variable X, then a ϕ MUST be inserted before each Basic Block Z that:

1. $\exists$ non-empty path $B \to^+ Z$
2. $\exists$ path from ENTRY to Z which does not go through B
3. Z is the FIRST node that satisfies i. and ii.

2 Mid Term Preparation2

2.1 LR(1) parsing

Build $LR(1)$ **Automaton:** An $LR(1)$ Item (i, a) is an extension of $LR(0)$ Item, where the next allowed Token a is considered. i is a $LR(0)$ item, a is an input Terminal, allowing Reduction using i when input is

[Step 1]: Define `CLOSURE()` to decide States.

```
set computeClosure(set I) {
    closure = I;
    do {
        for (every Item m in I) {
            /* Suppose m is A -> a.Bb, x here. */
            for (every Production Rule r: B -> c)
                for (every Terminal t in FIRST(b, x))    /* Including $ symbol. */
                    Add B -> .c, t into closure;
        }
    } while (there are updates in this iteration);
}
```

[Step 2]: Define `GOTO ()` to decide Transitions.

a

```
set computeGoto(set I, Symbol X) {
    result = {};
    for (every Item m in I) {
        /* Suppose m is A -> a.Xb, x here. */
        result = Union of result and CLOSURE({A -> aX.b, x});
    }
}
```

[Step 3]: Build $LR(1)$ Automaton. The dummy item here is $S' \rightarrow .S$, \$.

- Shorthand for $r, a_1; r, a_2; \ldots; r, a_n$ is $r, a_1/a_2/\ldots/a_n$
- A State will contain $A \rightarrow \alpha., a_1/a_2/\ldots/a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq$ `FOLLOW (A)`

Implementing $LR(1)$ Parsing

By constructing $LR(1)$ *Action* & *Goto* Table, we can achieve $LR(1)$ Bottom-Up Parsing similarly.

```
/* Create Action Table. */
void createActionTable() {
    for (every State Ii in Automata) {
        for (every input Terminal a) {
            for (each Item r in Ii) {
                if (r is A -> B.aC, x)      /* Shift is not effected. */
                    Add "shift GOTO(i, a)" in Action[i, a];
                else if (r is A -> D., a)   /* Reduce only when match. */
                    Add "reduce A -> D" in Action[i, a];
                else if (r is S' -> S., "$")
                    Add "ACCEPT" in Action[i, "$"];
            }
        }
    }
}
```

- May still leave Conflicts; If *no Conflicts happen*, then G is a $LR(1)$ Grammar

2.2 LALR(1) parsing

Build $LALR(1)$ Automata

A **Core** is the set of all $LR(0)$ Items in a $LR(1)$ State, ignoring the following Terminal symbol.

$LALR(1)$ *merges* all the $LR(1)$ states with the same *Core*.

- Is a *Trade-off* between Grammar range ($LR(1)$) v.s. Efficiency ($SLR(1)$)
    - Number of States in $LALR(1)$ Automata $=$ Number of States in $SLR(1)$ Automata
    - Will only introduce *Reduce / Reduce Conflict*s into original $LR(1)$ Parser; If *no Conflicts happen*, then G is a $LALR(1)$ Grammar
- Used in "YACC/Bison"

Other Issues for Parsers

Conflict Resolution

Conflicts cannot be 100% removed in LR Parsing; Also, *Ambiguous* Grammars are sometimes more human-readable. The possible solutions are:

1. Use context informations from Symbol Table
2. Always in favor of *Shift*
3. Use *Precedence & Associativity*, e.g.
    - $E + E$, met $+$, do Reduce since $+$ is left-associative
    - $E + E$, met $*$, do Shift since $*$ has higher precedence
    - $E * E$, met $+$, do Reduce since $*$ has higher precedence
    - $E * E$, met $*$, do Reduce since $*$ is left-associative
4. Grammar Rewriting

Context-sensitive v.s. Context-free

NOT Context-free Language $=$ CANNOT write a CFG for this Language.

- e.g. $\{\omega c\omega : \omega \in L((a+b)^*)\}$

Context sensitive grammar analysis is widely used in Intra-process analysis.