بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term static single-assignment. Figure 6.13 shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables p *and* q *in the SSA representation.*

(a) Three-address code.	(b) Static single-assignment form.
p = a + b	p_1 = a + b
q = p - c	q_1 = p_1 - c
p = q * d	p_2 = q_1 * d
p = e - p	p_3 = e - p_2
q = p + q	q_2 = p_3 + q_1

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

has two control-flow paths in which the variable x *gets defined. If we use different names for* x *in the true part and the false part of the conditional statement, then which name should we use in the assignment* $\mathrm{y} = \mathrm{x} * \mathrm{a}$*?*

Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the ϕ-function to combine the two definitions of $\mathtt{x}$*:*

$$\mathtt{if\ (\ flag\ )\ x_1\ =\ -1;\ else\ x_2\ =\ 1;}$$
$$\mathtt{x_3\ =\ }\phi(\mathtt{x_1, x_2});$$

Here, $\phi(\mathtt{x_1}, \mathtt{x_2})$ *has the value* $\mathtt{x_1}$ *if the control flow passes through the true part of the conditional and the value* $\mathtt{x_2}$ *if the control flow passes through the false part. That is to say, the* ϕ*-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the* ϕ*-function.*

In SSA form, each variable can only be assigned to once. So if a piece of code makes an assignment to a particular variable and then that variable is assigned to again, the second assignment must be modified to use a new variable. For example, the code

$$\begin{aligned} \mathtt{x} &= \mathtt{a} + \mathtt{b} \\ \mathtt{y} &= \mathtt{x} + 1 \\ \mathtt{x} &= \mathtt{b} + \mathtt{c} \\ \mathtt{z} &= \mathtt{x} + \mathtt{y} \end{aligned}$$

is transformed to

$$\begin{aligned} \mathtt{x}_1 &= \mathtt{a} + \mathtt{b} \\ \mathtt{y}_1 &= \mathtt{x}_1 + 1 \\ \mathtt{x}_2 &= \mathtt{b} + \mathtt{c} \\ \mathtt{z}_1 &= \mathtt{x}_2 + \mathtt{y}_1 \end{aligned}$$

In the original code, $\mathtt{x}$ *is assigned to twice and hence the introduction of* $\mathtt{x}_1$ *and* $\mathtt{x}_2$*. This representation makes it clear that, for example, the use of* $\mathtt{x}_2$ *in line 4 corresponds to its definition in line 3. That use on line 4 has nothing to do with the definition on line 1. But this protocol causes a problem. Consider the code*

$$\begin{array}{l} \mathtt{x} = 1; \\ \mathtt{if}(\mathtt{a} < 0)\ \mathtt{x} = 0; \\ \mathtt{b} = \mathtt{x}; \end{array}$$

which can be translated into a three-address code version:

$$\begin{array}{l} \mathtt{x} = 1 \\ \mathtt{if\ a} >= 0\ \mathtt{goto\ l1} \\ \mathtt{x} = 0 \\ \mathtt{l1}: \\ \mathtt{b} = \mathtt{x} \end{array}$$

But when translated into SSA form

$$
\begin{aligned}
&\mathtt{x}_1 = 1\\
&\mathtt{if\ a} >= 0\ \mathtt{goto\ l1}\\
&\mathtt{x}_2 = 0\\
&\mathtt{l1}:\\
&\mathtt{b}_1 = ???
\end{aligned}
$$

the value to be assigned to $\mathtt{b}_1$ *is going to be either* $\mathtt{x}_1$ *or* $\mathtt{x}_2$ *and it is only at runtime when the decision can be made. To resolve this issue, SSA allows us to write*

$$
\begin{aligned}
&\mathtt{x}_1 = 1\\
&\mathtt{if\ a} >= 0\ \mathtt{goto\ l1}\\
&\mathtt{x}_2 = 0\\
&\mathtt{l1}:\\
&\mathtt{b}_1 = \phi(\mathtt{x}_1, \mathtt{x}_2)
\end{aligned}
$$

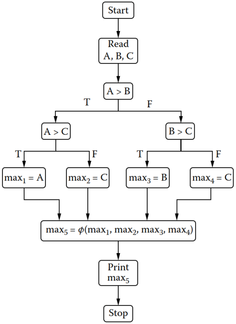
The ϕ-function is a function to indicate the meeting of control flow paths and the result is chosen according to which variable definition (the most dominating) was made in the control flow path most recently executed. At first sight implementing the ϕ-function appears difficult. But it can be implemented during code generation by ensuring that all variable parameters of the ϕ-function share the same register or storage location or by ensuring that appropriate register copies are made. All this additional complexity is justified by the fact that the data dependence information embedded in SSA is necessary for the implementation of a wide range of optimizations. For example, constant propagation is made very easy. If, say, variable $\mathtt{t}_{19}$ is found to have the value 3, then replacing all uses of $\mathtt{t}_{19}$ by 3 and deleting the definition of $\mathtt{t}_{19}$ is easily done. The generation of SSA is not trivial but good algorithms are now available.

A program is in SSA form if each of its variables has exactly one definition, which implies that each use of a variable is reached by exactly one definition. The control flow remains the same as in a traditional (non-SSA) program. A special merge operator, denoted ϕ, is used for the selection of values in join nodes.

Usually, compilers construct a control flow graph representation of a program first and then convert it to SSA form.

Read A, B, C
if $(A > B)$
  if $(A > C)$ $max = A$
  else $max = C$
else if $(B > C)$ $max = B$
    else $max = C$
Print max

Read A, B, C
if $(A > B)$
  if $(A > C)$ $max_1 = A$
  else $max_2 = C$
else if $(B > C)$ $max_3 = B$
    else $max_4 = C$
$max_5 = \phi(max_1, max_2, max_3, max_4)$
Print max_5



فصل ۱۹ کتاب اپل به‌کلی به فرم SSA اختصاص دارد

19

Static Single-Assignment Form

dom-i-nate: to exert the supreme determining or guiding influence on

Webster's Dictionary

INTERMEDIATE REPRESENTATIONS IN ACTUAL USE

☞ *In practice, compilers use a variety of IRs. Legendary FORTRAN compilers of yore, such as IBM's FORTRAN H compilers, used a combination of quadruples and control-flow graphs to represent the code for optimization. Since FORTRAN H was written in FORTRAN, it held the IR in an array.*

☞ *For years, GCC relied on a very low-level IR, called register transfer language (RTL). GCC has since moved to a series of IRs. The parsers initially produce a language-specific, near-source tree. The compiler then lowers that tree to a second IR, GIMPLE, which includes a language-independent tree-like structure for control-flow constructs and three-address code for expressions and assignments. Much of GCC's optimizer uses GIMPLE; for example, GCC builds static single-assignment form (SSA) on top of GIMPLE. Ultimately, GCC translates GIMPLE into RTL for final optimization and code generation.*

☞ *The LLVM compiler uses a single low-level IR; in fact, the name LLVM stands for "low-level virtual machine." LLVM's IR is a linear three-address code. The IR is fully typed and has explicit support for array and structure addresses. It provides support for vector or SIMD data and operations. Scalar values are maintained in SSA form until the code reaches the compiler's back end. In LLVM environments that use GCC front ends, LLVM IR is produced by a pass that performs GIMPLE-to-LLVM translation.*

☞ *The Open64 compiler, an open-source compiler for the IA-64 architecture, used a family of five related IRs, called WHIRL. The parser produced a near-source-level WHIRL. Subsequent phases of the compiler introduced more detail to the WHIRL code, lowering the level of abstraction toward the actual machine code. This scheme let the compiler tailor the level of abstraction to the various optimizations that it applied to IR.*

لینک‌های بسیار مفید:

https://gcc.gnu.org/onlinedocs/gccint/RTL.html

https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

https://llvm.org/

https://llvm.org/docs/LangRef.html

5.3 Applications of Syntax-Directed Translation

The syntax-directed translation techniques in Chapter 5 are applied in Chapter 6 to type checking and intermediate-code generation.

The application we describe here is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. (Chapter 6 also discusses approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.)

در قسمت ۵.۳.۱ از کتاب، دو SDD متفاوت برای ساخت AST معرفی می‌شوند

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

As discussed in Section 2.8.2, each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression $E_1 + E_2$ *has label* $+$ *and two children representing the subexpressions* E_1 *and* E_2*.*

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an op field that is the label of the node. The objects will have additional fields as follows:

☞ ***If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function $Leaf(op, val)$ creates a leaf object. Alternatively, if nodes are viewed as records, then $Leaf$ returns a pointer to a new record for a leaf.***

☞ ***If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function $Node$ takes two or more arguments: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field op and k additional fields for the k children $c_1, c_2, \ldots, c_k$.***

Example 5.11: The S-attributed definition in Fig. 5.10 constructs syntax trees for a simple expression grammar involving only the binary operators + and −. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with $'+'$ for *op* and two children, $E_1.node$ and $T.node$, for the subexpressions. The second production has a similar rule.

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

For production 3, $E \rightarrow T$, no node is created, since $E.node$ is the same as $T.node$. Similarly, no node is created for production 4, $T \rightarrow (\ E\ )$. The value of $T.node$ is the same as $E.node$, since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two T-productions have a single terminal on the right. We use the constructor *Leaf* to create a suitable node, which becomes the value of $T.node$.

Figure 5.11 shows the construction of a syntax tree for the input $a - 4 + c$. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of $E.node$ and $T.node$; each line points to the appropriate syntax-tree node.

At the bottom we see leaves for a, 4 and c, constructed by *Leaf*. We suppose that the lexical value **id**.*entry* points into the symbol table, and the lexical value **num**.*val* is the numerical value of a constant. These leaves, or pointers to them, become the value of $T.node$ at the three parse-tree nodes labeled T, according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for a is also the value of $E.node$ for the leftmost E in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for $-$ with the third leaf.

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 5.12 ends with p_5 pointing to the root of the constructed syntax tree. □

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

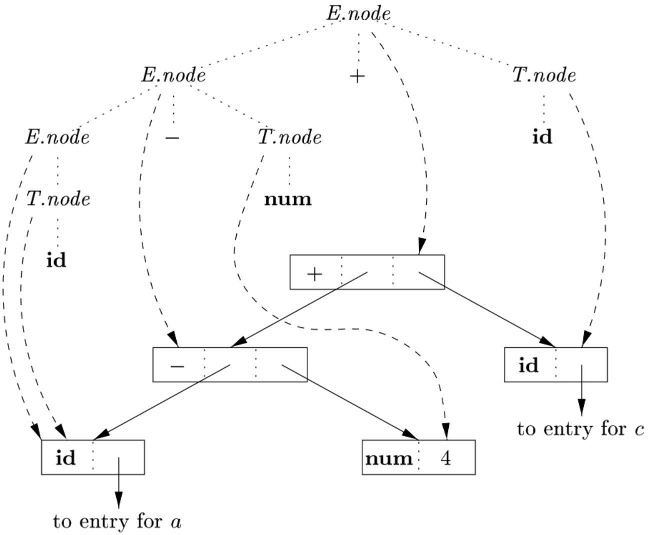


Figure 5.11: Syntax tree for $a - 4 + c$

$$
\begin{array}{ll}
1) & p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a}); \\
2) & p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4); \\
3) & p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2); \\
4) & p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c}); \\
5) & p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);
\end{array}
$$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

Example 5.12: The L-attributed definition in Fig. 5.13 performs the same translation as the S-attributed definition in Fig. 5.10. The attributes for the grammar symbols E, T, **id**, and **num** are as discussed in Example 5.11.

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5.3. In the desk-calculator example, a term $x * y$ was evaluated by passing x as an inherited attribute, since x and $* y$ appeared in different portions of the parse tree. Here, the idea is to build a syntax tree for $x + y$ by passing x as an inherited attribute, since x and $+ y$ appear in different subtrees. Nonterminal E' is the counterpart of nonterminal T' in Example 5.3. Compare the dependency graph for $a - 4 + c$ in Fig. 5.14 with that for $3 * 5$ in Fig. 5.7.

Nonterminal E' has an inherited attribute *inh* and a synthesized attribute *syn*. Attribute $E'.inh$ represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for E'. At node 5 in the dependency graph in Fig. 5.14, $E'.inh$ denotes the root of the partial syntax tree for the identifier a; that is, the leaf for a. At node 6, $E'.inh$ denotes the root for the partial syntax tree for the input $a - 4$. At node 9, $E'.inh$ denotes the syntax tree for $a - 4 + c$.

Since there is no more input, at node 9, $E'.inh$ points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of $E.node$. Specifically, the attribute value at node 10 is defined by the rule $E'.syn = E'.inh$ associated with the production $E' \to \epsilon$. The attribute value at node 11 is defined by the rule $E'.syn = E'_1.syn$ associated with production 2 in Fig. 5.13. Similar rules define the attribute values at nodes 12 and 13. □

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \mathbf{new}\ \ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \mathbf{new}\ \ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \mathbf{id}$	$T.node = \mathbf{new}\ \ Leaf(\mathbf{id}, \mathbf{id}.entry)$
7)	$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new}\ \ Leaf(\mathbf{num}, \mathbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

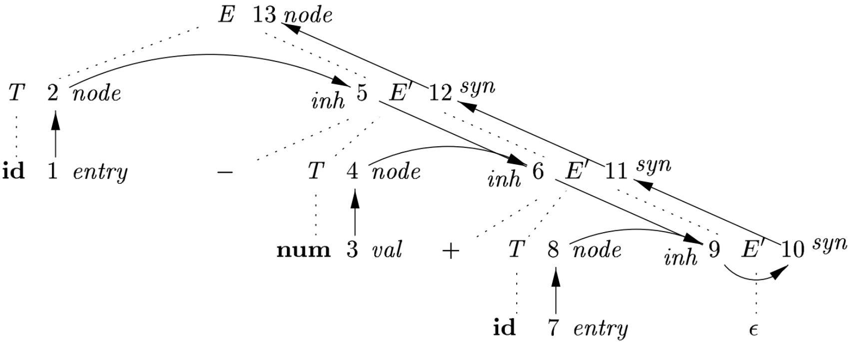


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13